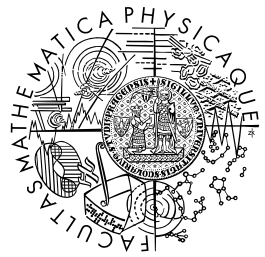

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Vojtěch Kloda

Convergent Real-Time Collaborative Document Editor and Repository

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	doc. RNDr. Martin Kruliš, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and software development

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to express my sincerest gratitude to my supervisor, doc. RNDr. Martin Kruliš, Ph.D., for his support, insight, and guidance throughout the course of this thesis.

I would also like to thank my friends who supported me, especially Jan Papesch and Tomáš Raunig, who provided valuable feedback on the many iterations of the algorithms that were developed.

Title: Convergent Real-Time Collaborative Document Editor and Repository

Author: Vojtěch Kloda

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Martin Kruliš, Ph.D., Department of Distributed and Dependable Systems

Abstract: Online document editors are useful tools that allow users to create, edit, and often also store and share documents. Some editors additionally support collaborative real-time editing for multiple users, allowing them to get live feedback from others.

Collaborative editors need to face the issue of conflicting user changes. To avoid desynchronization, such editors need to be able to resolve conflicts in a consistent way at the site of each user. To do so, editors use the technology of operational transformation, which proposes a series of complex algorithms aiming to achieve consistency.

The goal of this thesis is to develop a collaborative editor and repository prototype which is able to automatically resolve conflicting user changes and support large numbers of active users. To achieve this, a new theory for handling consistency is devised from an existing one by introducing a set of constraints to improve scalability.

Keywords: editor, document, collaborative, distributed, synchronization

Contents

Introduction

People want to share their ideas, and it is helpful to get live feedback. Whether it is a work or school assignment, a pet project, or a task list; if it is to be shared and viewed, it should be done conveniently. And if it is to be edited collaboratively, it should be done efficiently. There are many real-time editing applications that tackle this problem, but most of them have at least one of the following drawbacks.

1. Enforced authentication, which requires all users first to create an account to use the application. This hinders the speed of sharing documents with parties who have not used the software before.

2. The user rights for a given project cannot be changed, allowing anyone to make changes. In some scenarios, it is better for some users to be unable to change the content of a specific project.

3. Low performance, limiting the number of participants in sessions and reducing the responsiveness of the application.

4. Low concurrency support, causing conflicting changes to destabilize the document editing session, desynchronizing users, and making them resolve conflicts manually (usually done by one user forcing their document state on all other participants).

5. No long-term storage options for documents.

The prototype text editor introduced in this thesis attempts to avoid these problems while remaining simple and convenient to use. The need for authentication and the way documents are shared are determined by the owner of the documents, not the application. Performance is achieved by minimizing the size of transactions and using lock-free algorithms. Concurrency is handled by using well-known algorithms and adapting them to the needs of the application.

The drawback is the need for a dedicated server to host the application. Therefore, the server initialization process is designed to be as simple as possible.

Chapter overview

Chapter *1 Analysis* lists the use cases, functional and non-functional requirements, related work, an introduction to the problem domain, and an overview of the changes made compared to the source material.

Chapter *2 Consistency Model Design* explains the inner workings of the software, especially that of the algorithms handling concurrent editing.

Chapter *3 Implementation* describes the architecture of the system and its implementation.

Chapter *4 Evaluation* goes through the use cases and requirements defined in *Chapter 1* and describes whether or not they were met. Benchmarks of the software are also listed here.

Appendix *A* provides an in-depth formal description of the theory behind the software, which was heavily inspired by an existing article about concurrency but has many differences. The theorems introduced in this Appendix do not generally have a counterpart in the source article, therefore proofs were devised to show their correctness.

Appendix *B* describes how the prototype should be deployed.

Chapter 1

Analysis

1.1 Use Cases

This section will go over the anticipated use cases for the software.

1.1.1 Demonstration of New Programming Concepts

A programming teacher explained a new concept and wants to demonstrate it to his students in code. To do so, he opens an online document editor, creates a new project, and shares it with the students so that they can connect to it and see what he does in real time. The teacher does not want the students to make changes to the project, so he makes sure that they have read-only rights. The teacher then proceeds to create new files and folders, in which he demonstrates the concepts.

This scenario can be decomposed into the following actions.

1. The teacher creates a project in the editor.
2. The teacher shares the project with his students.
3. The students can see the contents of the project.
4. The students cannot change anything inside the project.
5. The teacher creates files and folders within the project.
6. The teacher edits the files.
7. Whenever a file is edited, the changes made are sent to the students.
8. The editor clients of the students receive the changes and display them.
9. The students see the changes in the order in which they are happening.

1.1.2 Programming Collaboratively

A group of programmers decides to work on a program collaboratively so that they can get live feedback from others. One programmer uploads the relevant part of the codebase to an online text editor and invites the others to join. After they connect, they proceed to add new functionalities to the code. They make sure that no one else can connect so that they are not disturbed.

The decomposition of the scenario follows.

1. A programmer copies files and folders to the editor.
2. The programmer then invites the others to join him in the editor.
3. The other programmers connect and can see the files and folders.
4. The participants edit files and see the changes of others in real time.
5. Whenever the programmers make potentially conflicting changes at the same time, the editor automatically merges them.
6. The programmers add new files and folders whenever necessary.
7. No one else is able to connect and view the files.

1.1.3 Sharing Code Samples

Some select programmers in an organization are tasked with maintaining code samples that others can use. They create a shared repository and organize the samples in a system of folders. They make sure that they are the only ones that can change the contents of the repository so that the others cannot introduce mistakes in the samples. Finally, they share the repository with the organization.

This scenario consists of the following actions.

1. The programmers create a repository.
2. The programmers create a system of folders.
3. The folders are filled with code samples.
4. A link to the repository is created that allows anyone to view it.
5. People who access the repository using the link cannot modify it.
6. The programmers later add more folders and samples.
7. Whenever the programmers feel that the name of a file or folder could be better, they can rename it.

1.2 Functional Requirements

1.2.1 Workspace Management

Workspaces are the topmost data structures encountered in the application. Each contains a system of documents and folders, a list of users that have access, and other metadata. To edit a document, a user first has to access a workspace and navigate to the document. The requirements for the management of workspaces follow.

1. Creation and deletion: Authenticated users can create workspaces and, by doing so, become their owners. Workspace owners can also delete their workspaces.

2. Basic file operations: Users with permission can create, delete, and rename documents and folders in workspaces.

3. Sharing: Workspaces can be shared with other users using a link or by directly permitting access to a registered user, displaying the workspace in the user's workspace list. Link sharing is the only way for an unregistered user to access a workspace.

4. Accessibility: Users with permission can edit the access rights of registered users, as well as their roles. Authorized users can also disable link sharing.

1.2.2 User Roles

If all clients had the same permissions in a workspace, its management would become more and more difficult with the growing number of clients, because different clients might have different ideas of how the workspace should be organized. Some clients might even have nefarious intentions to delete important files or provide access to users who should not have it.

Therefore, workspaces should support user roles that dictate which actions users can perform. The following workspace actions should be separated into different roles.

- Viewing the contents of a workspace.

- Editing the contents of a workspace by editing documents and performing file operations.

- Managing workspace aspects such as accessibility and roles of other users.

1.2.3 Concurrency

To improve the collaborative aspects of workspaces, users should be able to edit the same document concurrently. Additionally, users should see the changes of others in real-time to be able to react to them immediately and avoid producing conflicting content.

The system orchestrating the collaborative aspects should not be visible to the users; everything should happen automatically. In case the system fails and the users desynchronize, users should be notified immediately so that the lowest amount of progress is lost. If this situation arises, users should be able to solve the problem directly in the workspace without the need to reconnect.

1.2.4 Authentication

The workspace should support user authentication. Authenticated users should be able to view what workspaces they have access to and interact with them to the degree their role permits.

In addition, users should remain authenticated for some time even after stopping using the software to prevent repeated authentication.

The system should also be able to delegate authentication requests to an external authentication provider.

1.3 Non-Functional Requirements

1.3.1 Response Time

When a user makes a change to a document, the client will send out the change, and after some time other clients will receive it and apply it to their version of the document. The time between the user making the change and the other one receiving it can be split into three parts.

1. The first is how long the client waits before sending out this change. This waiting is generally necessary to not overwhelm the system with many small changes. Although clients can send out a message whenever a user adds a single character, waiting a small time period for more changes to aggregate can lead to dramatically lower bandwidth requirements as the network overhead for each message becomes smaller relative to the size of the change.

2. The second is the time it takes to get the message from one client to the other. This time cannot be easily influenced, as it depends on how stable and fast the internet connection of the clients is.

3. The last part is the time it takes to apply the received change to the document. This depends on how the application process is implemented.

The program should have a customizable waiting period and minimize the time overhead of applying changes. The waiting period and the network overhead should be the majority of the time spent between a user making a change and another one receiving it.

1.3.2 Bandwidth Usage

Sending and receiving changes consumes network resources, more so if the workspace has many active users. Additionally, the loading of documents and performing file operations consumes resources too.

To ensure that even users with low network bandwidth can use the application, the overhead of the accompanying metadata structures sent alongside actual changes should be minimal and should not increase with the growing number of users.

1.4 Related Work

There are many programs with functionality similar to that of the prototype devised in this thesis. This section will describe three different text editors with collaborative features and compare them with the one proposed in this thesis. A list of the editors chosen follows.

1. Google Docs[1] is a free text editor for browsers developed by Google LLC. It allows users to create documents, store them, and edit them collaboratively with others. Changes made by users are propagated to others in real time. Additionally, users can see where the cursors of other users editing the same document are. It also supports text and document styling, tracking changes, document translation, and spell check for some languages, along with other features[2].

2. Replit[3] is a free integrated development environment (IDE) for browsers developed by Replit, Inc. As an IDE, it allows users to create projects called Repls that can be populated with files and folders. Users can edit files collaboratively in real time, and files can be compiled and executed in an integrated terminal.

[1] Google Docs about page: https://www.google.com/docs/about/
[2] Google Docs features: https://support.google.com/docs/topic/1361462
[3] Replit homepage: https://replit.com/

3. Etherpad[4] is a free open-source self-hosted text editor for browsers developed by the Etherpad Foundation. To use Etherpad, one must first download the Etherpad backend and host it or connect to an already hosted service. Etherpad is available on GitHub[5] under the Apache License, Version 2.0. Similarly to Google Docs, Etherpad supports text and document styling, but additionally, Etherpad supports plugins that can extend its base functionality.

These examples were chosen because of their popularity and similarity to the one proposed in this thesis.

1.4.1 Comparison with the Proposed Editor

This section will present different features of the introduced editors and compare them with the prototype.

Algorithms

The algorithms of Google Docs and Replit are not known publicly; however, Google published a series of blogs, some of which give a high-level description of the concurrency algorithms used in Google Docs[1][2].

Based on the low level of detail in the blogs, their approach is identical to the one used in this thesis, with the exception that Google Docs only sends one change from a client to the server at a time and waits for a confirmation before sending another one. The reasoning behind this was not present in the blogs.

Similarly, the EasySync[3] algorithm used by Etherpad can also be mapped to the algorithm used in this thesis, as far as their description of it goes; however, as with the Google blogs, the description is not detailed enough to discern any finer differences.

Accessibility and Permissions

Similarly to the proposed editor, the three example editors support link sharing, which can be toggled. However, only Etherpad can allow unregistered users access to documents.

Replit handles roles through their *Teams* and *Organization* features[6], within which users can create shared projects. These roles are strictly managerial, allowing certain team members to create projects, delete them, add new users to the team, etc.

[4]Etherpad homepage: https://etherpad.org/

[5]Etherpad GitHub repository: https://github.com/ether/etherpad-lite

[6]Replit roles specification: https://docs.replit.com/organizations/roles-and-permissions

Standalone projects can be public or private, but making projects private and thus inaccessible to others is a paid feature. When viewing a project of another user, one cannot change it without being invited by its owner. However, the project can be forked.

The prototype editor does not support any grouping of users or project forking. However, it is possible to handle roles on a per-project basis, instead of doing so globally for a team of users.

Unlike Replit projects, which are public by default, their counterparts in the prototype are private. They can be easily made public by allowing link sharing.

Google Docs allows one to share documents from Google Drive. Documents can be publicly shared using a link or only with a selection of users.

There are three user roles; two are read-only with one allowing one to make comments, and the third allowing one to edit the document. Both public sharing and sharing with specific users support the three roles.

The prototype does not support document commenting; however, the read-only and edit mode for links is supported.

1.4.2 Similarity

The prototype is closest to Replit and Etherpad.

The purpose of the prototype is to aid software developers by providing features such as document storage inside folders, syntax highlighting, and the decomposition of files into a set of potentially endless rows instead of paragraphs; all of these features are shared with Replit. However, the prototype does not have an integrated terminal.

Unlike Replit and similar to Etherpad, the prototype is open-source. Users can deploy their own instance on their server or connect to an already existing one.

1.5 Introduction to Operational Transformation

Operational transformation (OT) is the process of transforming user changes (known as operations) to suit the needs of the recipient in collaborative systems.

Because operations are not received by others instantaneously, once they arrive, the document might be already in a different and incompatible state. However, these situations can be amended by transforming incoming operations.

Due to the complex nature of operational transformations, several *consistency models* have been devised that dictate how the problem should be approached.

This section will discuss what consistency model was chosen for this thesis and describe it.

1.5.1 Consistency Models

Collaborative text editors need to make sure that users see documents in their correct forms; in other words, they need to be consistent. Several theories have been developed to tackle the problem of consistency by proposing various consistency models. Consistency models define sets of restrictions for the system and claim that adhering to them will make the system consistent. However, many theories have been shown to have problems.

One of the first consistency models was the CC model[4] which defined the *casuality preservation* and *convergence* properties, which described how operations should be ordered and that documents should converge to one identical version over time.

Years later, the state-of-the-art CCI model[5] was proposed, which additionally defined the *intention preservation* property, which requires the system to preserve the syntactical effect of operations.

However, later on, it was discovered that the theory behind the CCI model was flawed; situations that led to non-consistent behavior were found.

The latest models attempted to improve the CCI model by redefining its properties to make it easier to introduce correctness proofs. An example would be the CR model[6].

Although the theory behind the CCI model was shown to be flawed, it was still chosen as the main source of inspiration for this thesis. The main reason was clarity; the theory provided the most in-depth explanations for the concepts it used, with an emphasis on how the system should be implemented for real-world usage.

Due to the high complexity of operational transformation, it was decided to prefer comprehensibility over correctness. However, the flaws of the original theory behind the CCI model were not ignored. This thesis proposes a new theory that still uses the CCI model, but changes many aspects of the original theory to mitigate its flaws and achieve better usability in real applications, such as reduced client bandwidth requirements, operation grouping, and enhanced intention preservation.

Because the article introducing the CCI model[5] will be referenced many times in this thesis, it will henceforth be known simply as "the article".

1.5.2 The CCI Model

The *article* proposes a *Consistency model*, called the *CCI model*, consisting of three properties: *Causality preservation*, *Convergence*, and *Intention preservation*. In order for a collaborative editing system to adhere to this model, it needs to have all three properties. These can be described in the following way.

1. Causality preservation: For any pair of operations O_A and O_B, if the effect of O_A was applied to the document where O_B was generated before its generation (i.e., the operation O_B is based on a document state that was changed by O_A), then O_A is applied before O_B at all client sites (O_B is dependent on O_A).

2. Convergence: When the same set of operations is applied to the document at all client sites, all copies of the shared document are identical.

3. Intention preservation: For any operation *O*, the effects of applying *O* at all sites are the same as the *intention* of *O*, and the effect of applying *O* does not change the effects of *independent* operations, where the intention of an operation is its desired syntactical effect.

1.5.3 Causality Preservation

The *article* uses the peer-to-peer network architecture, and thus there is no central authority that could provide information about the causal ordering of operations.

To achieve causality preservation, the *article* uses operation timestamps called *state vectors*. These state vectors are better known as *Vector clocks*[7] in the literature, which are extensions of *Lamport timestamps*[8] that are used in distributed systems to determine a partial ordering of events and to detect causality violations.

The state vectors are maintained by each client separately and sent along with each operation. Internally, the state vector contains an entry for each participating client, where each entry denotes the number of operations received from a specific client. They describe in what *context* the operation was generated and are used to determine how it should be transformed to be applicable to the document.

1.5.4 Convergence

Convergence is achieved by defining a *total ordering* algorithm that ensures that all operations are applied in the same order. An operation with a lower sum of its state vector is ordered before an operation with a bigger sum, and, in the case of equality, the unique numeric identifiers of the authoring clients are compared.

With causality and convergence achieved, the *Undo/Do/Redo Scheme* defined in the *article* can be utilized to apply incoming operations. The algorithm first undoes all operations that are totally ordered after the received operation, then applies the received operation, and lastly redoes the undone operations.

1.5.5 Intention Preservation

The last property, intention preservation, is much harder to achieve, and thus most of the *article* is dedicated to it. To achieve intention preservation, an operation that is about to be applied in the Undo/Do/Redo Scheme has to be first transformed to compensate for the changes made by other operations before it. The transformation is done by *include* and *exclude* transformations, which add or remove dependencies from the context of an operation, respectively. The goal is for the transformed operation to have all operations that are totally ordered before it in its context, making the operation aware of all previous changes. The transformation is executed by *The Generic Operation Transformation Control Algorithm* (GOTCA), which is able to transform any operation to its correct form, using a series of include and exclude transformations.

Include and Exclude Transformations

There are many types of include and exclude transformations, one for each ordered pair of operation types. The *article* defines two types of operations, *Insert* and *Delete*, which insert or remove text at a certain position in the document, respectively. Therefore, there are four include transformations and four exclude transformations (insert/insert, insert/delete, delete/insert, delete/delete). The effects of the transformations vary based on the specific parameters of the participating operations; some examples are changing the position of an operation, the length of its deletion range, or even splitting the operation into two.

Transformation Side-Effects

There are three special outcomes that include and exclude transformations can have. These are *Lost information* (LI), *Relative addressing* (RA), and *Operation splitting*.

1. Lost information occurs when the transformation reduces the amount of information the original operation had, making it impossible to be reversed. This is mitigated by saving the original form of the operation so that the transformation can be reversed in the future.

2. Relative addressing occurs when the position of an operation is dependent on the position of another operation, making undoing and redoing the second operation result in the first having a potentially incorrect position. This is mitigated by saving information about the operation on which another one is dependent with its position.

3. Operations can split during transformation into two. This can happen in several scenarios, one being a client deleting some text, and another inserting new text into the deletion range of the first before the Delete operation arrives. If the Insert operation is totally ordered before the Delete operation, the Delete has to include the Insert into its context. However, the Delete must not remove the text added by the Insert, as that would violate the intention of the client to delete a certain range of characters, of which the text in the Insert operation was not part. The Delete, therefore, has to split into two; one Delete for the left part of the deletion range and one for the right part, with the Insert in the middle.

1.5.6 The Garbage Collection Scheme

The *article* also introduces a distributed garbage collection scheme, which is used to remove unnecessary operations from the operation buffers of clients. These buffers store received operations that may be used in some future transformations, but no single client can decide if an operation in the buffer can still be potentially useful. The Garbage collection scheme makes clients periodically broadcast their state vectors so that all clients know how many operations each client sent out and how many were not yet received. With this information, clients can determine which operations will from that moment on never be used in the Undo/Do/Redo scheme and delete them from their memory.

1.6 Improvements and Changes to the CCI Model

This section will go through the main changes and improvements made in this thesis compared to the *article*.

1.6.1 Network Architecture

The biggest difference is the change of the network architecture to client-server. This has several major impacts on the underlying algorithms, as well as the required bandwidth of clients.

Clients in the peer-to-peer architecture have to broadcast their operations to all other clients, and the sizes of accompanying state vectors are based on the number of collaborating clients, resulting in the required client upload speed growing quadratically with regard to the number of clients.

The client-server architecture allows to replace the state vectors with a different timestamping data structure, called *operation metadata*, which consists of only four numbers. Clients also only send their operations once to the server, and

the server will then, in turn, broadcast them. This means that the required client upload speed is constant with respect to the number of clients. The required upload speed of the server will be quadratic, the same complexity as a client in the peer-to-peer approach, as the server receives operations from all clients and has to broadcast each one of them.

The client-server architecture allows using protocols such as WebSocket, which guarantee that sent operations will be received in the same order. This trivializes the problem of causality preservation, as operations from a certain client will be always received in the correct order and can always be applied (the reasoning for this will be explained in Chapter A); therefore, operation application will never have to be postponed, as can be the case in the *article*.

1.6.2 Total Ordering

The reduction of state vectors to operation metadata makes the original total ordering algorithm inapplicable. Therefore, a new algorithm was devised. Because the original algorithm was established based on the sum of state vectors, it gave priority to operations created earlier, even if they arrived much later. A client with a significant latency could thus impair the whole editing session, as his operations would be ordered before a potentially large set of already received operations made by clients with lower latency. The delayed changes could no longer have any meaning in the current document context, making them potentially disruptive.

The new total ordering algorithm is based on the time the server received an operation. This causes the operations sent by clients with high latency to have lower priority, causing them to be transformed to the new context instead of transforming a much larger number of operations with lower latency.

1.6.3 Operation Chains

The thesis also makes improvements to intention preservation. If a client, in the system proposed by the *article*, sends out two or more consecutive operations without receiving any, these operations could effectively be aggregated and applied as a single operation because they only depend on a single document state and on each other. However, there is no guarantee that the proposed total ordering algorithm will order these chained operations consecutively.

This can have the effect that some of these operations are transformed differently than others in the chain, leading to fragmented text additions (the full explanation of this phenomenon is detailed in Section A.3, with a high-level example in Section 2.2.3). This had been amended with the concept of *operation chains* and adding a new rule to the total ordering algorithm, which groups these chains together.

1.6.4 Two-Dimensional Documents

Another major feature is the expansion of operation position attributes. While the *Insert* and *Delete* operations with a single position attribute are sufficient, the lack of a more fine-grained position system results in much more complicated transformations. Because there is only one position attribute (describing at which character position should the operation take place), the addition of an Insert operation into the context of another operation situated after it will always result in a change of its position, i.e., the operation positioned lower in the document gets moved whenever an operation in the upper part gets added to its context. Whenever at least two people edit the same document, this will happen even if they edit completely unrelated parts of it.

This can be avoided by introducing a new position attribute and thus making the document text two-dimensional. The first dimension could be a row, a paragraph, a chapter, or any other unit of length (depending on the required granularity), and the second dimension would be the character position inside it. For the purposes of this thesis, only rows will be considered, as any other grouping will functionally behave the same.

Adding Insert or Delete operations into the context of another operation on a different row will no longer have any effect, greatly reducing the number of arithmetic operations needed for transformations. The drawback of this approach is the need for two more types of operations, the addition and removal of rows. These could still be handled by Insert and Delete operations, effectively deleting the newline character, but the logic of adding and removing rows is that much different than simply adding or removing a character; therefore, adding two new operations will make the transformation algorithms much clearer.

The new set of operations is called *Add*, *Del*, *Newline*, and *Remline*, where Add and Del were renamed to not be confused with the original operations Insert and Delete, as Add and Del have one additional position parameter and cannot add or remove lines, and Remline being an abbreviation of 'Remove Newline'. However, the introduction of new operations considerably increases the number of transformation algorithms needed, as one has to exist for each ordered pair, both as an include and exclude transformation. This results in 16 include and 16 exclude transformations, as opposed to the 4 and 4 of the two-operation approach.

1.6.5 Operation Grouping

The system proposed in the *article* did not allow clients to group multiple operations in a single message. Allowing this would enable a group of operations to share metadata, save bandwidth, and reduce the general network overhead of sending multiple messages.

For this reason, a buffer is introduced to store changes made to the document over a specified time period. After the period elapsed, all buffered changes are converted into a single aggregate operation with a single operation metadata structure and sent to the server.

Chapter 2

Consistency Model Design

This chapter will provide a high-level overview of the concurrency model that was developed, but will not go into detail on how the underlying theory works.

The theory used in the *article* was not used as the basis for the implementation, as the improvements and changes this thesis attempts to achieve would make its adaptation difficult and inefficient.

For that purpose, a new theory was devised. It keeps the same high-level approach to operational transformation, but the basis, such as how is total ordering established and how are operations structured internally, was completely remade. The formal description of the theory can be found in Appendix A.

This chapter will only discuss concepts related to consistency management. Other functionalities, such as file operations and user management, will be detailed in the next chapter.

2.1 Communication

The software uses the client-server network architecture. Clients send changes made by users to the server, and the server broadcasts them to all clients connected to the document.

The most notable benefit of the client-server architecture is that the metadata in operations sent out by clients is of constant size, compared to the peer-to-peer architecture used in the *article*, where the metadata grows linearly with the number of participating clients. Because the clients receive changes only from one source, the server, it is possible to devise a total ordering algorithm relying on metadata of constant size. The details of the devised total ordering algorithm can be found in Section A.3; however, a high-level description will also be present in this chapter in Section 2.2.3.

The following is a summary of the roles of the clients and the server.

2.1.1 Client

The role of the client is to display documents to users, capture user changes, and send them to the server.

The changes made by the users are represented by a structure called a *dif*. A dif describes how the document changed relative to the previous document state; for example, a dif could contain information to delete five characters at a certain position and replace them with the word 'sample'.

Before a dif is sent to the server, it needs to be enriched with metadata describing the previous document state and information about the authoring client. These metadata are necessary to correctly derive how the dif should be applied to the documents of other clients. The combination of a dif and such metadata is called an *operation*.

The client does not send out operations whenever the user makes a change to the document. Instead, once the user makes a change, the client will listen for potentially more changes for a short period of time, called the *buffering interval*. After the buffering interval ends, the client merges all the captured changes into a dif, generates the accompanying metadata, and sends them as an operation to the server.

2.1.2 Server

The role of the server is to broadcast all the operations received to all clients. The server sends operations even to their authors, who use them for synchronization purposes.

The server also sends the current document state and metadata needed by the transformation algorithms to newly joined clients. To be able to do so, the server needs its own copy of the current document state. Therefore, the server has to process all incoming operations as well and apply the changes to its working copy of the document.

The server is also responsible for periodically invoking the *Garbage collection scheme*. This collaborative event cleans the data structures used by the transformation algorithms on the server and on all clients.

2.2 Achieving Consistency

Section 1.5.1 introduced consistency models as sets of requirements that need to be adhered to for the system to be consistent. The CCI model was chosen for this thesis; therefore, the requirements of causality preservation, convergence, and intention preservation, listed in Section 1.5.2, need to be fulfilled.

This section will go over how these requirements were met.

2.2.1 Achieving Causality Preservation

The *article* achieved causality preservation by using state vectors, which could be used to derive on what operations a particular one depends on.

Each entry of the state vector describes on how many operations from a specific client an operation with this state vector depends. Once the recipient received the correct number of operations specified by the state vector sent alongside an operation, the operation could be applied; it would be causally ready.

The problem of causality preservation is trivialized in this thesis by using the client-server network architecture and communication protocols that guarantee that the order of messages is kept.

It can be shown that, in such scenarios, all the operations on which one might be dependent were already received by the recipient, and thus the application of operations does not have to be postponed. All received operations are automatically causally ready. Additionally, it can be shown that the four numbers present in operation metadata are sufficient to derive exactly on what operations another one depends. Appendix A formalizes these properties in Theorems 1 and 5 and includes their proofs.

2.2.2 Achieving Convergence

Causality preservation dictates whether or not an operation can be applied to the document. However, it does not mention when it should be applied.

Total Ordering

In case two clients had identical document versions and both users using said clients would make changes to the document at the same time, the other clients, which received both operations, would not know in what order they should be applied. If they chose arbitrarily, their document versions would likely diverge, because operation application is not commutative.

Consider a situation where all clients have the same empty document. If one user would write the word *cat* and a second one would write *dog* at the same time, the other clients could either end up with a document saying *catdog* or *dogcat*, based on which operation was applied first.

This could be solved by defining a total order of all operations. If all operations were applied in the same order at all client sites, their documents would be identical, because nothing could cause a discrepancy. Both the *article* and the thesis define an algorithm that determines the total order of operations.

Applying Out-of-Order Operations

Generally, it is not possible to know the total ordering of all operations before they are all received. Thus, when an operation is received by a client, it is possible that the client already applied an operation that is totally ordered after it because the client could not have known that an operation with a lower total ordering would arrive. The operation with a lower total ordering could no longer be applied to the document because the operation application needs to adhere to the total ordering.

To solve this, a scheme is devised that undoes operations from the document with a higher total ordering so that the received one can be applied, and then the undone operations are reapplied.

This scheme was defined in the *article* as the Undo/Do/Redo (UDR) scheme. This thesis adapts the UDR scheme to the new structure of operations; otherwise, it works the same.

2.2.3 Total Ordering Algorithm

This section will summarize how the *article* implements the total ordering algorithm, identify its issues, and describe how the thesis amended them in its implementation.

Total Ordering Introduced in the Article

The *article* uses state vectors to determine the total ordering of operations.

The algorithm is simple; an operation with a lower sum of its state vector is totally ordered before an operation with a higher sum. In case they have the same sum, the unique numeric identifiers of the clients are compared.

Although this algorithm can totally order any set of operations, it has several drawbacks.

1. The first is that some clients will always have priority over others because their client identifiers are lower. In case the sums of state vectors match, the client with the lower identifier will always have priority over the other.

2. The second is that clients that have received fewer operations in total have priority. This is because the state vectors they will be attaching to outgoing operations will have a lower sum, and thus will be totally ordered before other operations. Clients with higher latency would usually receive operations later; therefore, the higher the latency, the higher the priority.

The problem with this is that usually the longer it takes the operation to be received by others, the more outdated its content becomes. And such outdated content would have priority over more relevant operations.

3. The final shortcoming identified is that whenever a client sends out multiple operations in a row without receiving any, these operations have no special link between them, even though they could be treated as a single operation.

   Consider a scenario in which two clients write a sentence on an empty document (they both start writing at the beginning). Because the sentence is long and it takes more time to write, it is sent in multiple operations. However, the clients also have latency, so they do not see the changes made by the other one until they finish writing. Because the sentences were sent in multiple operations and because of how the total ordering algorithm is implemented, the total ordering would consist of interleaved operations of those clients. The resulting document would display the sentences interleaved with each other and make them incomprehensible.

Total Ordering in the New Theory

This thesis devises a new total ordering algorithm to solve the three identified issues. It does so by relying on the concepts of *Server ordering* and *Operation chains*, which are formally defined in Sections A.1.2 and A.3.

- Server ordering is the order in which the server receives and then transmits operations. The server transmits operations to clients in the order it receives them; therefore, using the fact that the communication protocol guarantees the order of send operations stays the same, all clients can derive the Server ordering by remembering in what order they received operations from the server.

- Operation chains represent sequences of operations sent by the same client that were not interrupted by the client receiving an operation from the server. The theory defines a simple mechanism for detecting them using only the operation metadata. Whenever the metadata of two operations differ only in the ID of the operations (meaning the ID of the author and the identifiers of the last operation received from the server are the same), these two operations are part of the same Operation chain.

If Server ordering were used to totally order all operations, it would solve the first two issues identified in the total ordering algorithm of the *article*.

The first issue of certain clients having a higher priority is solved by not considering the client identifiers at all. The server transmits operations without considering from which client they came.

The second issue of later operations having a higher priority is solved by having a single source of operations, the server. If it takes a long time for an operation to reach the server, the newer ones would already be transmitted by it and thus would be ordered before the late one.

However, Server ordering alone does not solve the third issue, the grouping of operations sent by the same client in an uninterrupted sequence. It is solved by ordering operations lexicographically; that is, primarily grouping operations together if they belong to the same Operation chain, and secondarily based on their Server ordering.

The total ordering algorithm used in this thesis orders operations in this lexicographic order.

2.2.4 Achieving Intention Preservation

This thesis uses the same approach to achieve intention preservation as the *article* (described in Section 1.5.5); that is, it uses a series of include and exclude transformations on received operations to transform them to fit the current document.

The thesis adapted the algorithm handling these include and exclude transformations, the GOTCA algorithm, to handle the new formats of operations. While the adapted GOTCA relies on the same concepts, the specific mechanisms vary greatly, as the changes in the new theory, namely, the new total ordering algorithm and aggregation of multiple changes in the same operation, have severe impacts on the low-level aspects of the algorithms.

This section will go over the high-level concepts of GOTCA, which hold for both the *article* version and the new one. The inner workings alongside the formal definition of GOTCA can be found in Appendix A, GOTCA is detailed in Section A.4.10.

Dependency

The essence of GOTCA is to transform an operation to be *dependent* on all operations totally ordered before it which are known to the client performing the GOTCA.

An operation O_B is *dependent* on operation O_A if O_B was generated from a document to which O_A was applied. If O_A was not applied to the document, O_B would be independent of O_A.

Whenever a client receives an operation, it does not have to be dependent on all operations that are totally ordered before it because the client that generated it could not yet have received all such operations.

Context

The *context* of an operation is the series of operations it depends on. If the operations present in the context of operation O were applied in the same order to an empty document, the resulting document would be the document from which O was generated. This holds because O is dependent on all operations that were applied to the document before O was generated and that document was the product of applying all these operations to an initially empty document.

An additional property of context is that the operations it contains are totally ordered. This is due to the convergence property, which states that the operations are applied in the defined total order.

GOTCA thus changes the context of an operation to include all operations that are totally ordered before it.

Inclusion and Exclusion Transformations

An analogy to adding an operation O_A to the context of an operation O_B would be to apply O_A to the document from which O_B was generated and then re-generating it. However, this approach would be extremely resource intensive.

Instead, include transformations were devised that can simulate this process using a series of simple instructions. These include transformations are able to append the context of an operation with a different operation. However, they are not able to insert it to an arbitrary position inside the context.

The context of an operation effectively behaves like a stack; to be able to add something to a specific position inside it, all elements that follow it need to be removed first, starting from the last.

Similarly to how include transformations append contexts, exclude transformations are able to remove the last entry. Exclude transformations act as the inverse of include transformations.

GOTCA

GOTCA orchestrates the process of how the context of an operation changes.

Returning to the stack analogy, GOTCA tries to change the stack representing the context of the input operation O to contain the same elements in the same order as the sequence of operations totally ordered before O. A straightforward way of constructing a stack that consists of a specific sequence is to empty the stack and then add all elements of the sequence in order. This approach would

again be unnecessarily resource intensive. If the stack has the same elements as the sequence up to some point, these stack entries do not have to be removed; only the remainder can be removed and the stack filled with the remaining elements in the sequence.

The stack analogy hides many intricacies faced in GOTCA, such as that in order to exclude an operation from a context, the exact form of the operation has to be known beforehand, which might not always be the case, and thus GOTCA also has to find the correct forms of such operations. However, these parts of the algorithm would be difficult to explain without first understanding the full theoretical background that leads to them; therefore, it is recommended for interested readers to go through it by reading Appendix A.

2.3 The Garbage Collection Scheme

As mentioned in Section 2.2.3, the clients derive the Server ordering by saving operations they received from the server as a sequence. Additionally, the GOTCA changes the contexts of operations by making them match the sequence of known operations that are totally ordered before them. The clients thus have data structures to save these sequences; the Server ordering buffer (SOB) holds the sequence of operations received from the server, and the History buffer (HB) holds the sequence of known operations in total order.

However, these data structures could grow arbitrarily large over time.

The execution time of algorithms like the GOTCA is dependent on the size of these data structures; as an example, the GOTCA transforms the context of an operation to match the sequence of operations totally ordered before it, which is a segment of the HB starting from the beginning and ending at some index.

The *article* introduced the Garbage collection (GC) scheme, which is able to remove entries from such data structures to reduce the execution times of algorithms that depend on them. Only the adaptation of the GC scheme to the theory devised in this thesis will be described, as the *article* did not define Server ordering and thus does not introduce the SOB.

Determining Removable Entries

The context of an operation O was defined as the series of all operations on which it is dependent. However, in Section 2.2.4, it was alluded that whenever the context of O starts with the same sequence of operations as the sequence of operations totally ordered before O (which is a segment of the HB), that part of the context does not need to be changed.

This implies that if the matching parts of the context and the series of totally

ordered operations were both removed, the GOTCA would not be affected. This is the principle of the GC scheme; it determines which parts of the HB and SOB would be redundant in all future executions of the algorithms and removes them.

To determine this, the GC scheme must know the contents of the HBs and SOBs of all clients.

Whenever a client generates a new operation from a document, it is dependent on all operations that were applied to it; these operations are its context. These operations are exactly the entries of the current HB of the client because the current state of the document was produced by applying operations in total order to it, and the HB holds exactly these operations in total order.

It can be shown that whenever the HBs of all clients start with the same operations, they will start with it at any point in the future as well; the intuitive reasoning for this is that all new operations will have to be totally ordered after this starting sequence of the HB.

Combining these facts implies that the contexts of all new operations will start with the same sequence, making it redundant. The GC scheme determines this sequence and proceeds to remove it from the HBs of all clients. It also finds the removed operations in the SOBs and removes them as well.

Execution of Garbage Collection Scheme

The GC scheme is triggered by the server because the clients should not have knowledge of the data structures of other clients.

It is triggered whenever a certain number of operations is sent by the server. While this approach could cause the scheme to be triggered even when nothing could be removed, devising a different trigger condition was deemed unnecessary, as the measured time complexity of executing the GC scheme was negligible compared to processing operations.

Once the GC scheme is triggered, the server starts collecting information about the HB and SOB data structures of the clients. Sending the whole HB and SOB to the server would consume too much bandwidth; therefore, clients only send a single number, the length of their SOB.

The server has its own version of the HB and SOB; by receiving only the length of the SOBs of clients, the server is able to simulate what the client data structures contain. Once the server receives this information from all clients, it is able to determine what data should be removed. The server then cleans its own data structures and sends a garbage collection command to all clients to clean their data structures.

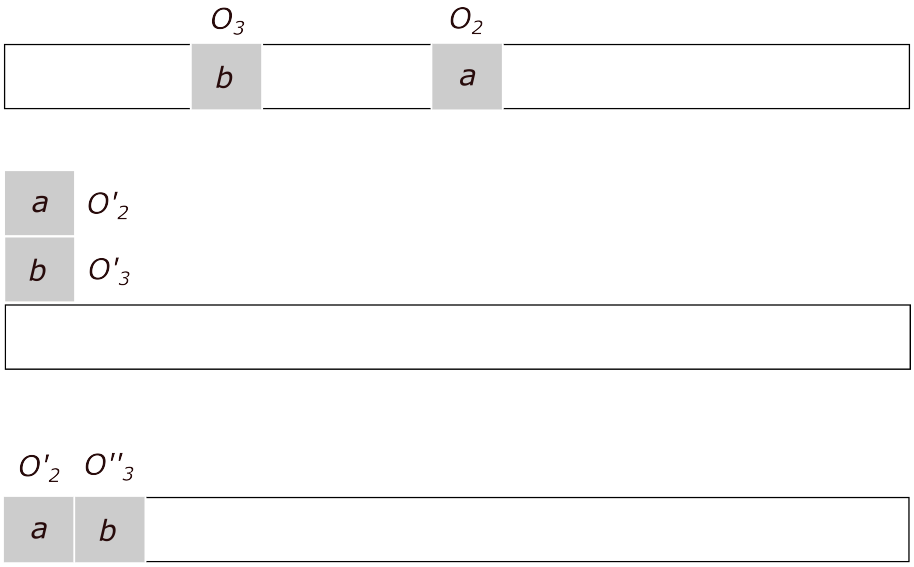

Figure 2.1 Depiction of the same-origin lost information side-effects problem. The illustration shows the document from the perspective of C_2. The individual parts reflect the document before, during, and after the execution of the UDR scheme.

2.4 Handling Desynchronization

Desynchronization occurs whenever two different clients apply an operation in a different way, making their document versions different and incompatible. This phenomenon is usually caused by an incorrect transformation result.

In this thesis, two issues identified in the *article* that could cause desynchronization were not solved. This section will introduce these issues and propose possible solutions.

2.4.1 The Same-Origin Lost Information Side-Effects Problem

An issue was identified that can result in two operations swapping places under certain conditions. Figure 2.1 provides an illustration of the situation.

Demonstration of the Problem

There are two clients, C_1 and C_2, and they are editing a shared document. C_1 deletes some text, sending out operation O_1, but before C_2 receives this operation, C_2 adds two characters separately to the range of text C_1 deleted. Both of these character insertions are sent out as different operations; O_2 and O_3. Additionally, the second character C_2 added in O_3 is placed before the first one in O_2; this is

crucial for the problem to occur. The server orders the operations so that the deletion operation O_1 is performed first, and then the two insertion operations O_2 and O_3. To further simplify the illustration, assume that C_1 deleted the entire document.

Once C_2 receives operation O_1, C_2 has to transform its operations using the UDR scheme, as they were ordered after it. The UDR scheme first undoes operations O_2 and O_3, then deletes the whole document by applying operation O_1, and finally redoes O_2 and O_3.

During the *undo* stage, the UDR scheme internally has to make the undone operations independent of each other. The *do* stage deletes the whole document; the undone operations now independently have to add the characters to the beginning of the empty document, as there is nowhere else to place them. This is illustrated in the second part of figure 2.1.

The *redo* stage makes the undone operations dependent again. It does so by including O_2 to O_3. The include transformation for two Insert operations with the same positions adds the length of the included operation to the transformed one. This results in O_2 being placed at the beginning of the document and O_3 behind it.

However, O_2 was originally placed behind O_3.

Cause of the Problem

When O_1 was applied to O_2 and O_3 in the *do* stage, both operations O_2 and O_3 lost information of where they were originally placed. This information is stored internally, but it is not used in this scenario, because retrieving lost information is reserved for transformations that act the opposite to those that caused the loss of operations.

The solution to this problem would be for all operations to consider lost information; however, it is not clear how this amendment should be approached, as the time when the loss occurred would have to be incorporated into the transformation algorithms.

2.4.2 The Multilevel Lost Information Problem

During the implementation of this thesis, it was discovered that operations that have already lost information are being transformed incorrectly if the transformation is to cause the operation to lose more information. This occurs because the data structure describing lost information, a *Wrap*, does not support multiple levels of lost information (see Definition 38 in Section A.4.5).

It is not known whether a simple change of redefining Wraps to hold a list of the structures necessary for describing lost information would mitigate this issue.

It could potentially also introduce new problems, such as reduced performance, increased memory usage, and increased algorithmic complexity.

2.4.3 General Approach to Desynchronization Resolution

The issue of desynchronization can be either approached by proving that it cannot happen in the system or by implementing active mitigation methods once it occurs. Because problems causing desynchronization were identified in this thesis, the latter approach had to be taken.

In order to mitigate desynchronization, it has to be first identified. Different approaches can be taken, such as periodically comparing document versions of clients that were transformed to some well-known state, verifying the results of individual transformations, or detecting failed operation applications. This thesis takes the last approach as it was the easiest to implement.

Whenever a failed operation application is detected, all clients receive a notification that desynchronization occurred. Clients then have the option to force their document version to the whole system. Once a client forces his version, all clients will receive a copy of it, resynchronizing the system.

Chapter 3

Implementation

In order to demonstrate the concepts introduced in Chapter 2, a collaborative text editor was implemented. This chapter will describe the overall architecture, design decisions, key components of the software, as well as how it was tested.

3.1 Architecture Overview

This section will list the core components of the system, provide a description of how they work and how they communicate with each other, and describe how the system could be extended. Figure 3.1 provides a high-level illustration of how the system is designed.

3.1.1 System Components

This section will first explain the thought process behind the choice of components, and then the specific components comprising the system will be listed.

Design Motivation

As stated in the previous chapters, the system uses the client-server network architecture. Therefore, the standard approach of dividing the system into a client, a server, and a database component could be taken.

However, it was decided to have two different servers handling requests rather than one, as the requests can be separated into two distinct categories; thus having a server handling each would create a better decomposition of the system.

The first server would handle requests that are stateless and usually resolved by simply fetching a static page or performing an operation on the database, such as verifying credentials. All of this would be unrelated to the collaborative aspects of the software.

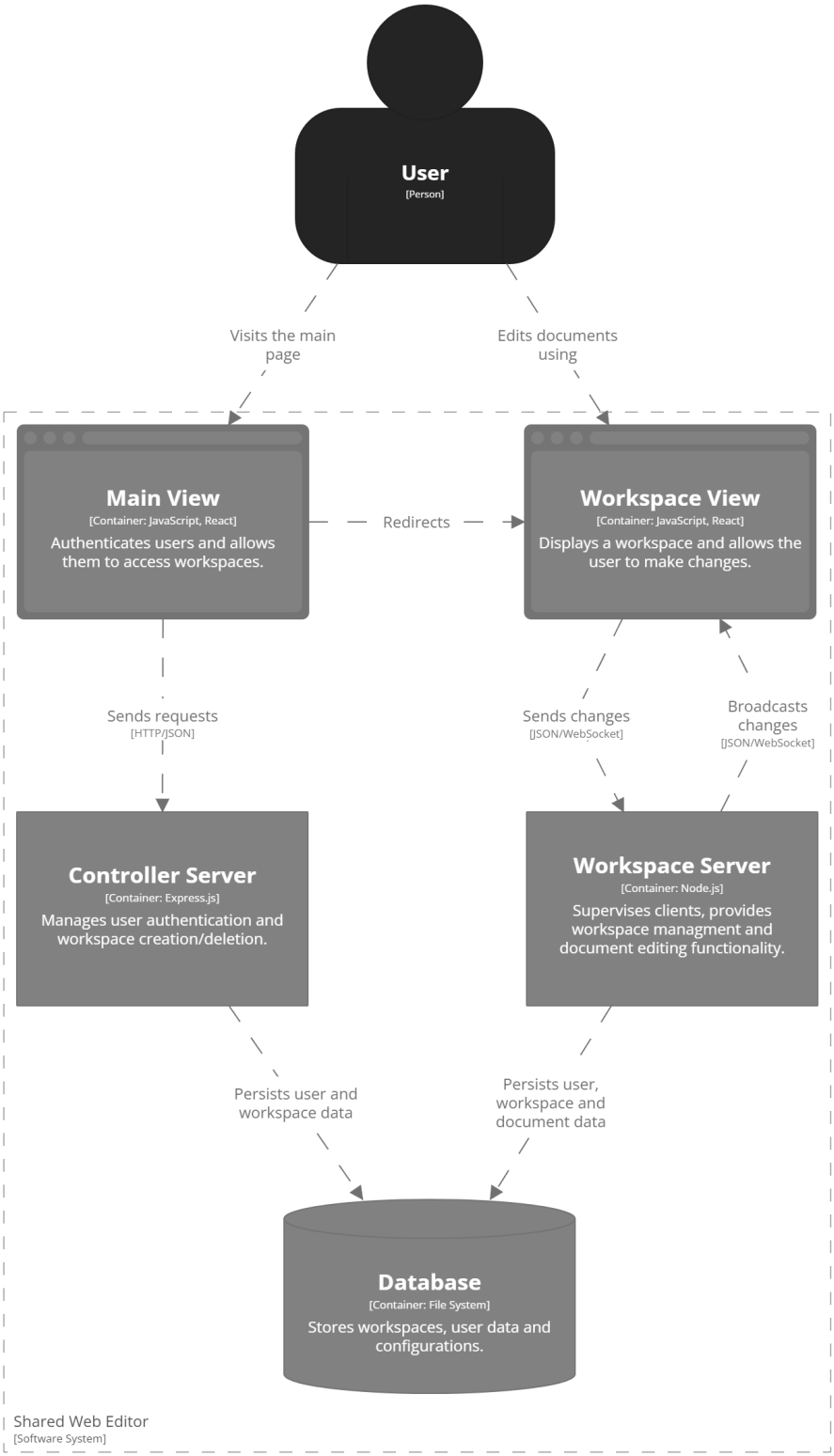

Figure 3.1 Diagram of the Shared Editor Architecture

The second server would handle stateful requests, all of which would concern collaboration. Examples would be the operations containing user changes, which are transmitted to all currently listening clients, as well as document requests, which besides fetching the current document subscribe the requester to all new changes occurring in it and making it participate in garbage collection events.

These servers would also have very little overlap, as one is mainly concerned with serving pages and the other with handling the logic of concurrent editing. This allows to split the server code into two parts, each of which could then leverage the benefits of different frameworks or programming languages altogether.

Components Overview

The editor is composed of four core components.

1. Controller server: A web server that serves static pages, authenticates users, provides access to workspaces to authorized clients, and handles workspace creation.

2. Workspace server: A server that accepts connections from clients and links them to a specific workspace. It handles the requests of clients, allowing them to manage the workspace and collaboratively edit documents.

3. Workspace client: A web client that displays workspaces and their contents, establishes connections to the Workspace server, captures user changes, and applies the changes made by other users.

4. Database: A minimalistic database that stores workspaces, user data, and configurations.

Figure 3.1 shows the components listed. The *Main View* illustrates the entry point of the software, which is provided by the Controller server. The *Workspace View* is a different view that is shown once the user requests access to a specific workspace. It contains the Workspace client, which handles the stateful interaction with the Workspace server.

3.1.2 Controller Server

The main responsibility of the Controller server is to serve static pages and expose endpoints for stateless requests.

Static Pages Served

There are four main pages Controller server serves; the *Login View*, the *Create Workspace View*, and the *Workspace List View*, which were aggregated into the *Main View* in figure 3.1, and lastly the *Workspace View*.

1. Login View: Allows users to log into the application by sending the login credentials to the Controller server.

2. Workspace List View: Lists all workspaces to which the user has access, along with the role the user has in that workspace. The user can interact with a workspace listing to access the workspace itself by being redirected to the Workspace View.

3. Workspace View: Downloads the Workspace client from the Controller server, which provides all the functionality to this view. After the Workspace client is downloaded, the Workspace View does not interact with the Controller server anymore; all further communication is directed to the Workspace server.

4. Create Workspace View: Allows users to create new workspaces. The only required input that the user is presented with is the name of the workspace; all subsequent management, such as adding clients to it and manipulating its file structure, is done in the Workspace View.

Exposed Endpoints

As mentioned in the Login View, the Controller server has an endpoint for authentication. This endpoint responds with a JSON Web Token (JWT) if the credentials were valid. The JWT is stored in the Cookies of the browser and is sent alongside every request for the Controller server so that the user does not have to authenticate again. The Controller server also has an endpoint for the removal of the JWT, so that users can log out from the application.

Additionally, the Controller server has an endpoint for creating workspaces.

3.1.3 Workspace Client and Server

The Workspace server handles the requests of the Workspace clients.

To simplify the workspace clients, it was decided that each can only connect to a single workspace. However, a user can view multiple different workspaces by opening additional browser tabs.

Roles

As stated in the user roles requirement (Section 1.2.2), the application should support a system of roles that separate certain workspace actions.

The role system was designed so that each user has an implicit role tied to each existing workspace which can be changed by administrators. This allows the application to determine whether a user has access to a workspace or not by checking the role of the user; a separate system for determining access does not have to be introduced.

The role system had been made hierarchical; a more permissive role allows all actions that a less permissive one can perform, in addition to new ones. It was reasoned that a more complex system of roles would not be necessary because of the low number of possible actions.

A list of all defined user roles follows.

1. None: If a workspace assigns this role to a user, the client cannot interact with the workspace in any capacity, not even display it. This is the default role that all users have.

2. Viewer: This role enables the user to view the workspace file structure and document content, but forbids any manipulation.

3. Editor: Editors can edit documents and thus can participate in collaborative editing.

4. Workspace Editor: Unlike editors, workspace editors can also manipulate the workspace file structure.

5. Admin: Admins can add users to workspaces, change their roles, and change the workspace access type.

6. Owner: The default role for the creator of the workspace. The owner's role cannot be changed. Compared to the Admin role, the Owner can assign the Admin role to others and delete the workspace.

The Editor and Workspace Editor roles were not merged into one, as it was reasoned that a workspace could potentially have a file structure of high importance that should not be changed, but users should still be able to edit its documents.

If a user has the role of Viewer or higher in a workspace, that workspace will be displayed in the Workspace List View.

Workspace Sharing and Access Types

Workspace access types define how a workspace can be accessed. There are three access types; the *Privileged* access type, and then the pair of *Everyone with link* and *Everyone with link (read-only)* access types. The *Privileged* access type allows only authenticated users to connect if their role allows. The remaining access types additionally allow unauthenticated users to join the workspace.

If *Everyone with link (read-only)* is selected, all unauthenticated users are treated as if they had the *Viewer* role. The *Everyone with link* access type gives unauthenticated users the *Editor* role.

Each workspace has a unique link assigned to it. If a user follows the link, he can view the workspace, given that his role is sufficient or that one of the link access types is used. While users added to a workspace can navigate to it using the Workspace List View, the workspace link is the only means of access for unauthenticated users.

Client-Server Communication

Once a connection is established, the Workspace client and server communicate with each other by exchanging a number of different message types. These types can be grouped into the following categories.

1. Initialization: The Workspace client wishes to establish a connection to the server. To do so, the client needs to send the identification of the workspace it wishes to connect to and an authorization JWT (in case the access type is *Privileged*). If the client is authorized to access the workspace, the server will send an object that describes the workspace file system, the role the user has in the workspace, and other metadata. While the user role is used to determine what features are available to the client, disabling or enabling certain UI elements, the server also validates all client requests. Once the client is initialized, it can perform any action defined by its role.

2. Document editing: All messages related to collaborative editing sessions. If the user wishes to view or edit a document, the client sends a document request to the server. The server responds with the document content along with all the data necessary for concurrent editing. As long as the user does not close the document, the server will send the client all operations related to the document, as well as garbage collection requests. A client can have multiple documents open.

3. Workspace management: Manipulation of the workspace file system, the addition of users to the workspace, and the change of the workspace access

type. These requests are handled similarly; the request is sent to the server, it is validated, the underlying data structures change, and a response detailing the change is broadcasted to all clients in the workspace.

The connection is closed either when an unauthorized client requests access to a workspace with the *Privileged* access type, or when the client is terminated.

3.1.4 Database

The main requirement for the database was to be as minimalistic as possible, preferring extendability over performance, as it was anticipated that its contents and structure will often change during development. For that reason, the database was devised as a system of files and folders.

Designing a database in this fashion made the storage of workspace file systems trivial, as the file systems could simply be stored in a subfolder inside the database.

However, the components responsible for interacting with the database were kept as simple as possible, defining the least possible amount of operations the database shall support, so that in case the system should be upgraded in the future, only these database components would be impacted.

The database components were made safe to use by identifying operations that could produce conflicts and using a locking mechanism to allow only one to take place at a time.

Structure

The database is divided into two main parts; the storage of workspaces and the storage of user data.

Because workspaces already have a structured hierarchy of files and folders, their structure is kept in the database. They are also accompanied by metadata files, which describe their structure and users with access to them.

The data stored about users are minimal. It is limited to a unique identifier, the credentials of the user, and a list of workspaces the user has access to.

3.2 Design Decisions

This section will cover what technologies were used to develop the prototype and how it was tested. Key parts of the software will also be highlighted and possible future extensions will be discussed.

3.2.1 Programming Languages

This section will go over the choices of programming languages for the individual components of the system. The reason why a specific language was chosen will be defended and possible alternatives will be listed.

Workspace Client

Because the Workspace client will run in the browser, JavaScript was chosen for its implementation, as JavaScript is a common choice for web applications with many available frameworks and libraries to ease development.

Alternatives to JavaScript include different JavaScript flavors, like TypeScript or CoffeeScript, but these were not chosen because of the lack of experience with these languages.

WebAssembly could have been used for the more computationally intensive parts of the client to increase performance, but it was decided to spend time improving the core concepts of the resource-intensive algorithms rather than optimizing a specific version.

Workspace Server

The workspace server runs on the backend, thus many different languages could have been chosen. Originally, the initial version of the server was written in JavaScript so that both the Workspace client and server could share the same libraries for the algorithms developed.

However, it was later decided to rewrite the Workspace server in a language that has better support for parallelization, as many parts of it could be run in parallel.

C# was chosen because of its task-based asynchronous programming framework, which provided good parallelization support. Additionally, C# is garbage collected, which helps mitigate potential memory leaks.

For this reason, C++ was not chosen as it was reasoned that safe memory deallocation would introduce non-trivial complexity to the code.

Due to the lack of experience with other languages with good parallelization support, other alternatives were not considered. Additionally, it was reasoned that C# would not have any notable drawbacks during the implementation, making it a good choice.

Controller Server

The Controller server also runs on the backend. Because its main purpose is to serve static pages and expose several API endpoints, it was decided to implement

it in a language with a good framework for these use cases.

There are many alternatives with such frameworks, such as Python, JavaScript, or C#. In the end, it was decided to use JavaScript, as this would allow the Controller server to use parts of the user management library developed for the Workspace client.

3.2.2 Libraries

The project uses several libraries to delegate some functionalities to already existing software. This section will list the most important libraries used and discuss possible alternatives.

The views used in both the Workspace client and the Controller server are written in *React*[1], which is a popular choice for reusable user interface components. Because the views of the application are simplistic and do not contain any complicated logic, it would not be difficult to substitute it with a different interface library altogether. React was chosen only due to past experience with the framework.

The Controller server uses the *Express*[2] framework to serve static pages and expose endpoints. Express was chosen because it was the most popular JavaScript framework at the time of consideration.

The user interface elements that represent editable documents in clients come from the *Ace Editor*[3] library. Of the other editor libraries considered, Ace Editor provided the simplest API for the manipulation of specific rows and the capture of row-specific changes, which is important when creating and applying user changes that use a two-dimensional positioning system.

The other libraries considered were *CodeMirror*[4] and *CodeJar*[5].

3.2.3 Communication

This section will discuss what protocols and technologies were used for communication between the components of the system.

Workspace Clients and Server

The Workspace clients and server use the *WebSocket* protocol for communication, as their communication is inherently stateful, and WebSocket guarantees that the

[1] Official website of React: https://react.dev/

[2] Official website of the Express framework: https://expressjs.com/

[3] GitHub repository of the Ace Editor: https://github.com/ajaxorg/ace

[4] Official website of CodeMirror: https://codemirror.net/

[5] GitHub repository of CodeJar: https://github.com/antonmedv/codejar

order of messages being sent is the same as the order they are received, which is important for the synchronization algorithms used. It also offers low overhead, which is useful in situations where many users connect to the software and edit documents. Additionally, WebSocket is supported by most browsers.

The *HTTP* protocol could not be used effectively, as the communication between Workspace clients and the server has to be bidirectional. This could be solved if the clients periodically queried the server for updates; however, this would only introduce drawbacks in the form of potentially unnecessary requests without providing any benefit.

Another alternative would be *WebRTC*[6], but this technology is used primarily for peer-to-peer communication, which does not occur in the system.

Controller Server

The Controller server was designed as a RESTful server which exposes a few simple endpoints. These endpoints do not return any structured data; they return static pages or data in JSON format with at most two key-value pairs, as they are designed to execute commands like user authentication or workspace creation rather than returning complex data structures. Due to this, the Controller server does not support detailed data queries using query languages like *GraphQL*[7], because they would not provide any benefits.

The Controller server communicates over HTTP, which is a straightforward option for the uses it has to fulfill.

3.2.4 Key Parts of the Software

The software has many important parts that are essential for its proper execution. Only the Workspace client and server will be considered because most of the complexity lies with them.

All of the following parts and classes are present in both the client and server, with small differences between them. The duplicity is caused by the fact that both the client and the server need to transform operations and modify documents, as well as handle any requests that might modify the file system, either to visualize them to the user or to save them to the database. The server versions will be described first, followed by possible differences in the client.

[6]WebRTC homepage: https://webrtc.org/?hl=en
[7]GraphQL homepage: https://graphql.org/

Transformation Library

The largest and most essential part of the software is the transformation library. The library implements the theory defined in Appendix A. The intended usage of the library is to use only its method implementing the Undo/Do/Redo scheme, which applies a received operation to the local document and accompanying data structures. The said data structures do not need to be modified outside of using them in the Undo/Do/Redo scheme, with the exception of the garbage collection scheme, which is able to remove unnecessary entries from them. The data structures are also used when the client generates operations because the operations store information about the latest operation received from the server.

Workspace

An important part of the Workspace server is the *Workspace* class. This class contains handlers for all client messages and ensures that only authorized users can perform requested actions. Instead of handling all aspects of request execution, the Workspace class either delegates requests to other classes or calls the correct methods exposed by other classes, like the class handling the database (*Database*) or the one handling the structure of the file system of a given workspace (*FileStructure*). The Workspace class also governs the lifecycle of open documents, which are represented by the *DocumentInstance* class. It determines what documents should be instantiated, terminated, or to which a given operation should be delegated.

The client version does not handle any database interactions as it has no access to the database. Instead, it holds the logic of what components of the user interface are currently being displayed and what document the user sees.

Document

The DocumentInstance class holds the necessary data for a single document, as well as the logic for its manipulation. This class also enacts the garbage collection scheme by periodically checking the amount of garbage-collectible data on clients and sending garbage collection commands.

The client version is named *ManagedSession*, as it does not hold the document data directly but instead holds a *Session*, an object defined by the *Ace Editor* library that provides an abstraction of the document. However, the client version still manipulates the document in the same way the server version does, it only uses different means for it.

The client version also listens for any user changes and starts a listening window for them. It then creates an operation from the captured changes and sends it via the Workspace to the server.

File Structure

Instances of the FileStructure class act as in-memory representations of the actual documents and folders of a single workspace. It is used to reduce the database load and to provide faster responses to requests that do not need to actually modify the documents or folders themselves, such as if a document exists, if it can be renamed to a different name, or if a new document with a given name can be created. Due to this, the FileStructure needs to represent the proper state of the existing documents and folders, a task that is handled by the Workspace class. The FileStructure class also handles mapping document and folder paths to numeric identifiers so that client requests interacting with the file system do not need to contain lengthy file paths, but just a single number.

The clients do not have access to the database, so any changes to the actual files need to be received from the server. The client version also handles the graphical representation of the file system so that the user can interact with it.

3.3 Testing

The prototype was tested using unit tests that targeted the transformation library.

Unit tests were only devised for the transformation library, as the main focus of the development of this thesis was to devise and implement the new theory for the consistency model. The development of the theory and unit testing of its implementation was interleaved to make sure that the new concepts are correct and usable.

Many different approaches to unit testing were introduced over the time the prototype was being developed to make sure that the theory and its implementation is correct.

Include and Exclude Transformation Tests

The simplest type of unit tests devised, these tests target the low-level transformation functions. Each test scenario starts with an operation O_A, either includes or excludes operation O_B, and compares the result with the expected operation O_C.

The problem with these tests is that they are very time-consuming to write because there is an effectively endless amount of possible input operations. It is also not easy to determine what the expected result should be, as operations can contain multiple changes, each of which interacts with the others during the transformation. Additionally, transformation side-effects can occur (as defined in Section 1.5.5), which further complicate the test design.

A total of 131 unit tests of this kind were defined.

Identity Tests

To make writing many tests more time efficient, tests that take only one operation as input were devised.

As mentioned in the previous chapters, include and exclude transformations should be inverse functions. The intention of these tests is to show that excluding operations and then including them back results in the original operations.

To be able to devise these tests with only a single input operation, the test scenario internally splits the input operation into many, one for each individual change the original contained, and then performs the exclusion and inclusion transformations on them.

These tests do not necessarily show the correctness of the individual include and exclude transformations, because the input transformation could be flawed and still have an inverse function; however, it is likely that such errors would only be present in one of the two transformation functions.

A total of 105 identity tests were defined.

Autogenerated Identity Tests

To make writing tests even more time efficient, a script was developed that randomly generates operations and performs the identity test on them. Whenever the test fails for a specific operation, that operation is captured, serialized, and appended to a file holding all such failed test cases. A utility script was devised that tests all the failed operations again for convenience. Additionally, the generator is seeded, making the generated test cases deterministic.

The operation generator works by iteratively building an operation by making randomized changes to a hypothetical document. In each iteration, the generator decides what kind of change it wants to make (either adding text, removing text, or adding or removing lines), then decides where the change should occur with regards to the current document to make sure invalid changes are not made, and finally sets the severity of the change if possible (the number of characters added or removed).

During development, this testing approach was able to detect the most errors. Even though the quality of the average autogenerated test case might not be as high as for manually devised tests, this approach is able to cover many more execution paths, as manually devising a test case for each possible path would be close to impossible due to the large number of possible transformation combinations.

The generator was unable to produce new failed test cases after all errors found in the first 10 million were fixed. In total, over 1 billion test cases were autogenerated and executed during the course of development.

High-Level Transformation Algorithm Tests

These tests aim to detect possible flaws in the high-level transformation algorithms, namely the GOTCA and the UDR scheme.

The general approach taken in these tests was to first initialize the data structures used by these algorithms (the SOB and HB), devise some possible incoming operations from different clients, and finally let the algorithms process them.

These test cases were the hardest to devise as they involve many operations and additional data structures, and determining the correct results could take hours.

Once it became too difficult to create new meaningful test cases, an alternative approach was taken. An editing session with multiple clients was started under extreme conditions, mainly clients having several seconds of latency. The clients then proceeded to make exclusively conflicting changes, such as all of them adding and deleting text at the same place in the document. This was repeated until the system desynchronized. If desynchronization occurred, the server logs were inspected, and a test scenario was built from them. Because these scenarios were very complex due to large numbers of conflicting changes, they were simplified to isolate the cause of the desynchronization.

Although this approach is highly unsystematic, it provided many more quality test cases in a shorter time when compared to their manual creation. Both the unresolved issues described in Section 2.4 were detected using this approach.

A systematic way for creating test cases in this way was not devised because not all issues identified in the existing test cases were solved (all of the unresolved test cases were related to the two unresolved issues).

A total of 66 high-level tests were defined, with 5 of them failing.

3.4 Future Extensions

This section will list several possible extensions of the system.

Hosting on Different Machines

Both the Controller server and Workspace server have to run on the same machine because of how the database is currently implemented. However, redesigning the system to be able to work on multiple machines would only entail adding a network API to the database.

In case multiple instances of the Workspace server were needed, they could be hosted on multiple machines, given that all had access to the database and that

a load balancer was implemented, which would dictate what workspaces would be managed by a specific Workspace server.

Undoing Changes

The current system does not support undoing document changes directly in the client. However, undoing changes is implemented and used internally by the Undo/Do/Redo scheme to apply incoming user changes. The reason why this was not extended to clients is that the changes need to fulfill a set of special conditions in the UDR scheme, which do not have to be met in general.

Implementing the undo functionality for clients would thus require extending the theory.

Integrated Terminal

Because the prototype aims to help develop software, it would be useful if the users could execute their code and see whether it behaves correctly.

This feature could be added either by using an already existing library that allows to interpret code in the browser or by executing the code in a virtualized environment on the server and sending the results to the clients.

Chapter 4

Evaluation

4.1 Fulfillment of Use Cases

This section will go over the use cases defined in Section 1.1 and note whether the prototype fulfills them.

The individual evaluations will first briefly summarize a specific use case and then group the actions defined in it, noting for each group whether the prototype supports its actions.

4.1.1 Demonstration of New Programming Concepts

This use case described how a programming teacher wanted to demonstrate a new concept to his students by sharing an online editor project with them, in which he demonstrated it.

1, 5 The prototype allows to create workspaces that support basic file operations.

2, 3, 4 The teacher can share the workspace either with a link with read-only access or by adding all students to the workspace with the read-only role. In both cases, the students can view the contents of the workspace.

6, 7, 8 Whenever a Workspace client registers a change to a document, it automatically sends it to the Workspace server, which broadcasts it to all other participants. Received changes are automatically applied to the correct document.

9 The order of changes is kept due to the network protocol, which guarantees it, and because the Workspace server sends out changes in the order it receives them.

4.1.2 Programming Collaboratively

This use case involved a group of programmers that collaboratively added new functionalities to some code they uploaded to an online editor.

1 The prototype does not support directly uploading files and folders to the workspace, but it can be simulated by recreating the file structure inside it and copying file contents.

2, 3, 7 To share a workspace only with specific people, they need to be manually added one by one with their desired roles.

4, 5 The prototype supports concurrent editing, and thus is able to automatically resolve conflicting changes. In case the conflict resolution fails and the system desynchronizes, the prototype detects this event and provides active mitigation methods.

6 The prototype supports basic file operations for users with sufficient roles.

4.1.3 Sharing Code Samples

This use case concerned itself with a group of programmers that wanted to share code samples with an organization.

1, 2, 3 As described in the previous use case evaluations, workspaces support basic file operations. The code samples can be created by adding files to the workspace and adding the code samples inside them.

4, 5 The workspace can be shared using a link with read-only access. However, there is no mechanism that would allow only members of the organization to access the samples without adding each member manually. This can be resolved by hosting the editor on an internal network of the organization.

6, 7 Users can perform file operations as long as they have a sufficient role in the workspace.

4.2 Fulfillment of Requirements

This section will evaluate whether the prototype met the functional and non-functional requirements defined in Sections 1.2 and 1.3, respectively.

A list of all requirements and their evaluations follows.

Workspace Management The prototype supports the defined workspace actions. Details of how are provided in Section 3.1.3.

User Roles The prototype supports roles that separate the defined actions. Details are provided in Section 3.1.3.

Concurrency The prototype is unable to immediately detect desynchronization. Otherwise, this requirement is met.

The current desynchronization detection system works by checking whether the incoming user changes can be applied to the Workspace server document. When a change cannot be applied, a notification is sent out to clients, and the users can resolve this by forcing someone's document as the correct one, copying it across all clients, and synchronizing the system.

Whenever the transformation algorithms transform a user change incorrectly, it might still be a valid and applicable change. However, in practice, it was observed that changes made after an incorrectly transformed change was applied to the document tend to be incorrect as well, and thus the number of incorrect changes starts to increase dramatically. It usually does not take long for a change to produce an invalid document state.

It should be noted that the speed at which desynchronization is detected is derived from the number of changes being applied. If the changes were sent out seldomly, the system could potentially be desynchronized for long periods of time. However, the chance of an incorrect transformation raises with the frequency of changes, and therefore seldom changes are unlikely to result in desynchronization.

Authentication The prototype does not support using an external authentication provider. The remainder of the requirement was met as detailed in Section 3.1.2.

Response Time While it was not proven that this requirement was met, the benchmarks in the following section indicate that the Workspace server performs well enough even under extreme conditions, given then the clients buffer their changes for 200 ms before sending them to the Workspace server (this is also the default setting in the prototype).

Bandwidth Usage This requirement is met; the operation metadata structure consists of only four numbers as mentioned in Section 1.6.1 and further detailed in Section A.2.

4.3 Workspace Server Benchmarks

To gain a better understanding of how the system performs, tests were made to determine how the Workspace server performs.

The goal of these tests is to show that the Workspace server performs well during loads. Specifically, the time it takes to process a single operation is much less than the default Workspace client buffering interval, as defined in Section 1.3.1.

All of the tests were executed on a machine operating on Windows 10 (OS build 19044.3086) with an Intel Core i7-12700K[1] Processor, two Kingston FURY Beast DDR5[2] Memory sticks, and the ASUS PRIME Z690-P[3] motherboard with an integrated network card.

The tests were run from the Windows Subsystem for Linux (WSL) command-line interface (build 19041). The Workspace server was deployed in Docker (v20.10.17) using the .NET 6 Runtime base image.

4.3.1 Testing Strategy

It is difficult to get an accurate measurement of how the software generally performs because it requires simulating the usage patterns of real users. Therefore, the tests are much simpler; they use simulated Workspace clients that perform a simple action repeatedly.

The first step of the test is to start the Workspace server. Not the standard version of the Workspace server is used, but an augmented version for testing, which periodically prints performance measurements to a file. Otherwise, the Workspace server is identical.

The next step is to connect the simulated Workspace clients to the Workspace server. These clients only contain the logic for handling concurrent editing, the user interface elements were removed. Additionally, the clients do not apply the operations they receive from the server, as this would severely increase the stress on the system. However, they still properly manage the data structures that handle what metadata operations have and partake in the Garbage collection scheme just as standard Workspace clients would. These clients connect to the Workspace server using the standard WebSocket connection using a loopback address.

[1]CPU specification: https://www.intel.com/content/www/us/en/products/sku/134594/intel-core-i712700k-processor-25m-cache-up-to-5-00-ghz/specifications.html

[2]Memory specification: https://www.kingston.com/datasheets/KF548C38BBK2-32.pdf

[3]Motherboard specification: https://www.asus.com/motherboards-components/motherboards/prime/prime-z690-p/techspec/

Each test connects a specific number of clients to the Workspace server; a portion of them is active, and the remainder is passive. A periodic interval is defined after which all active clients send a single operation to the Workspace server, simulating the buffering interval of real clients.

The tests will measure the time it took to process an operation. This time is divided into two parts; the time it took to apply the operation to the document using the UDR scheme, and the time spent transmitting the operation to all clients. While more actions are done during operation processing, these actions take several orders of magnitude less time to execute, thus they were excluded from the measurements.

During the processing of an operation, an instance of the C# *Stopwatch* class is created to measure the time spent transmitting and a second one to measure operation application speed. The measurements are timestamped with the time since the program started and regularly appended to an output file.

4.3.2 Tests

This section lists several tests performed on the prototype. Each test will start with a description of its scenario followed by two or three plots, one detailing the time it took to apply one operation, and the second one detailing how long it took to transmit it to the clients; both in milliseconds. Each box in the plot aggregates the data measured in one minute.

To demonstrate how the Workspace server performs with many connected clients, the following scenarios connect 200 clients to the server. Of the 200 clients, 40 will be active and the remaining 160 will passively listen for changes. To simulate the default client buffering interval used in the prototype, the active clients will send out an operation every 200 ms. This totals 5 operations per client per second, or 200 operations per second in total. Because each operation is transmitted to all clients, of which there are 200, a total of 40000 operations are sent by the server each second on average.

Each test scenario was run for 20 minutes, resulting in an average of 240000 measurements of operation processing per test.

Scenario 1: Adding Text to the Same Position 1

In this scenario, every operation sent by the clients adds 5 characters to the beginning of the document. This simulates an average user typing speed of 25 characters per second. The average amount of characters added to the document per second is 1000.

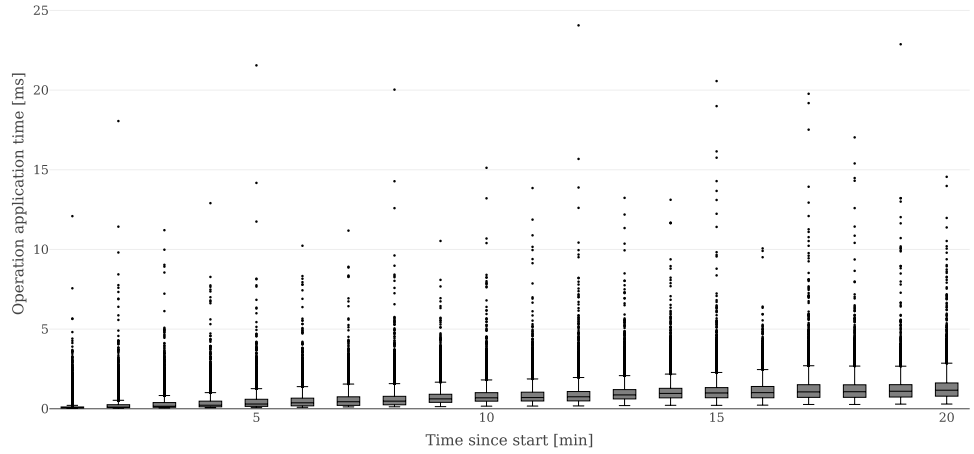

Figure 4.1 Box plot of operation application time where each operation consists of adding 5 characters to the beginning of the document.

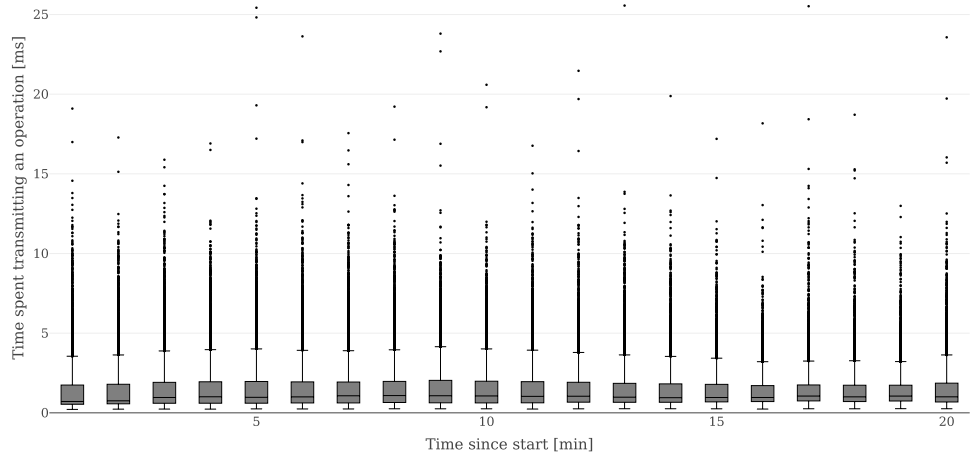

Figure 4.2 Box plot of the time it takes to transmit an operation to 200 clients.

Scenario 2: Adding Text to the Same Position 2

This scenario is the same as the previous one, but instead of adding 5 characters, each operation adds 20. This results in an average user typing speed of 100 characters per second and a total of 4000 characters added to the document on average per second.

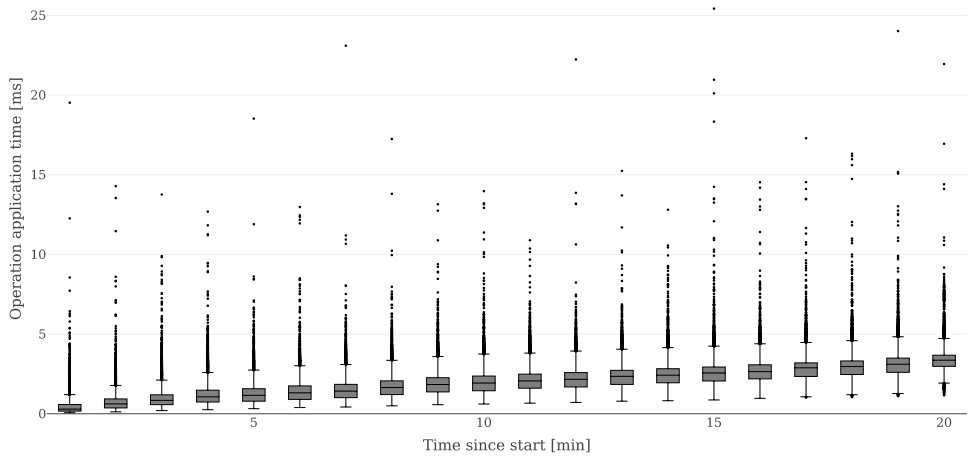

Figure 4.3 Box plot of operation application time where each operation consists of adding 20 characters to the beginning of the document.

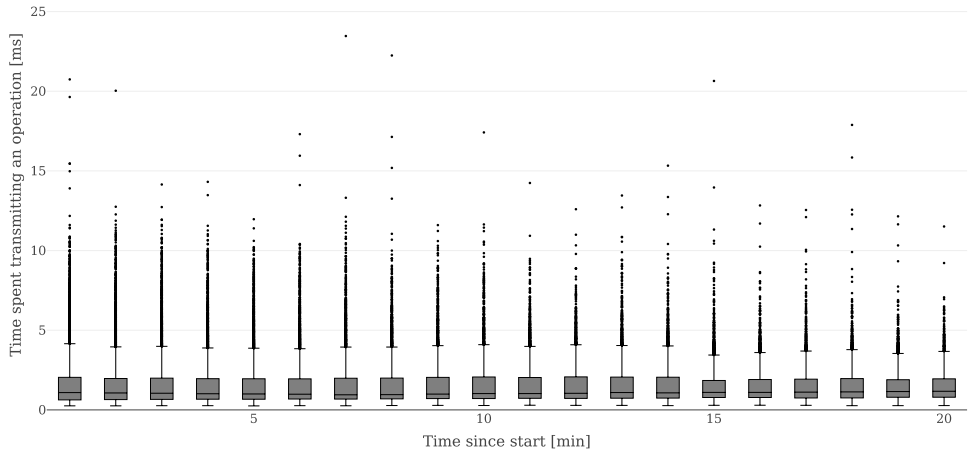

Figure 4.4 Box plot of the time it takes to transmit an operation to 200 clients.

Scenario 3: Adding Text to Different Rows

In this scenario, instead of the clients adding characters to the same position, each client has a unique row to which it adds the characters. The operations consist of 20 characters, the same as in the previous scenario.

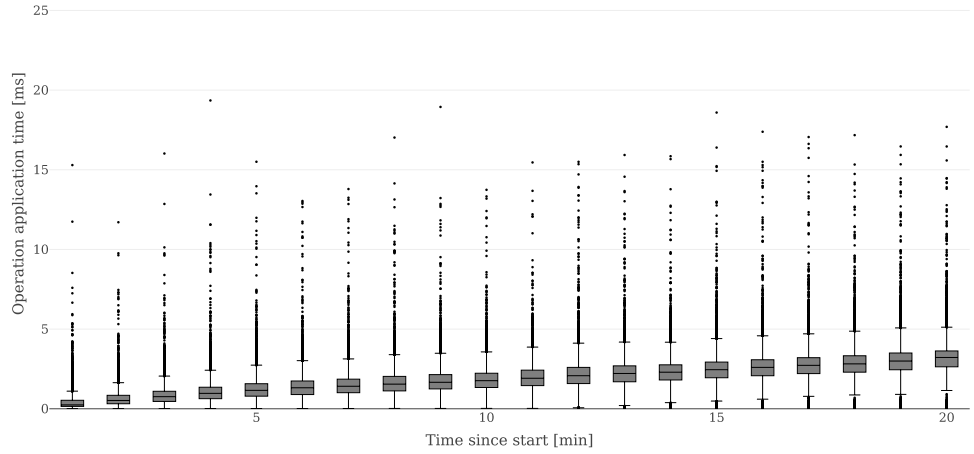

Figure 4.5 Box plot of operation application time where each operation consists of adding 20 characters to the beginning of the document.

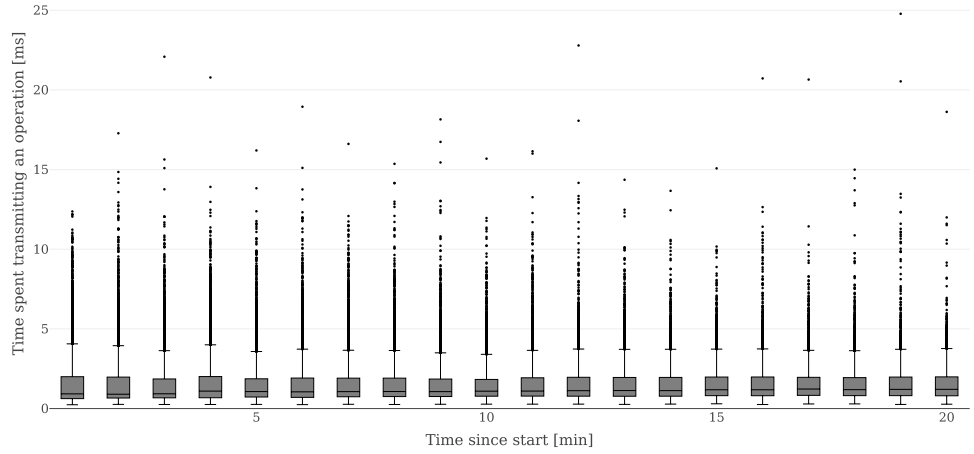

Figure 4.6 Box plot of the time it takes to transmit an operation to 200 clients.

Scenario 4: Adding and Deleting Text From the Same Position 1

In this scenario, there are two operation types that are sent by clients. The first is adding 20 characters to the beginning of the document. The second is removing 20 characters from the beginning. These operations are alternating; in every cycle, the clients either add or remove characters.

During the deletion cycle, all clients aim to remove the same 20 characters from the beginning of the document. In order to preserve their intentions, the

UDR scheme that is executed during operation application will only remove the first 20 characters once, as no client wished to remove characters positioned anywhere else.

This results in an average of 2000 characters being added to the document per second, and 50 deleted.

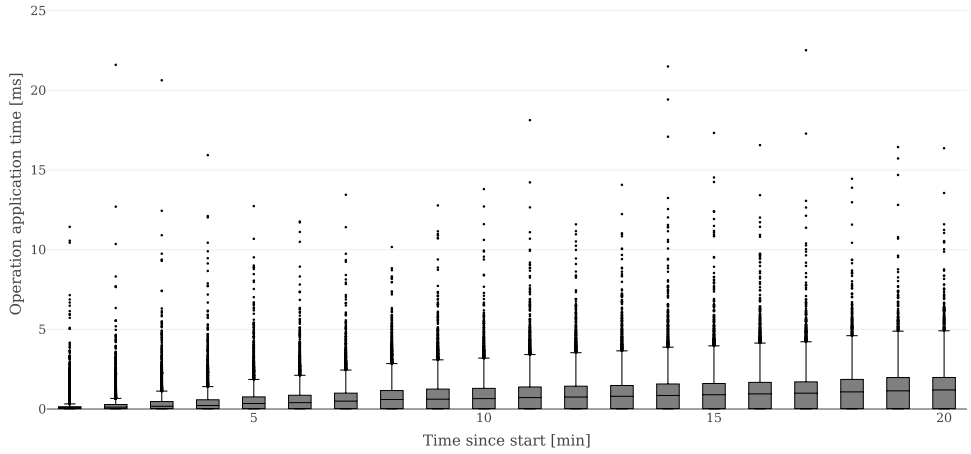

Figure 4.7 Box plot of operation application time where each operation consists of either adding or removing 20 characters from the beginning of the document.

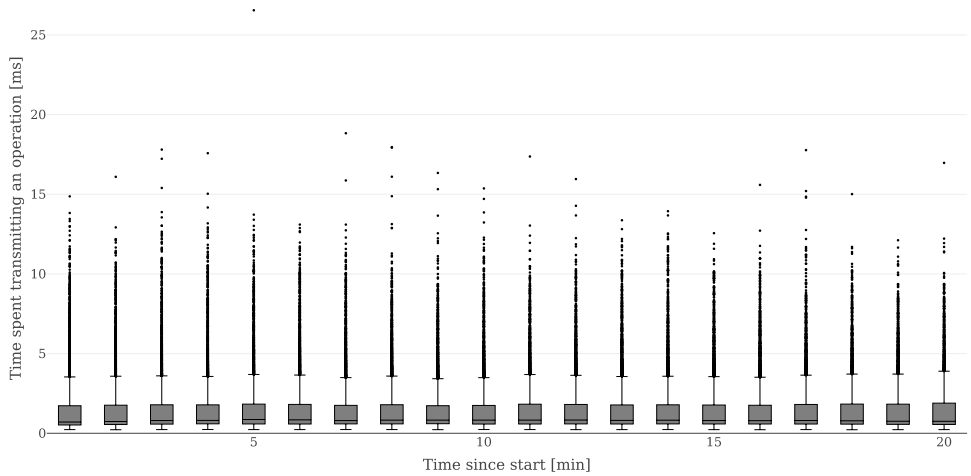

Figure 4.8 Box plot of the time it takes to transmit an operation to 200 clients.

Scenario 5: Adding and Deleting Text From the Same Position 2

This scenario sends the same operations as the last one, but instead of the deletion operations deleting the same text at the beginning of the document, they now delete different segments without overlap.

Compared to the previous scenario, an average of 2000 characters is being added to the document per second, and 2000 are deleted. Thus, unlike all the previous scenarios, the document in this one does not grow over time.

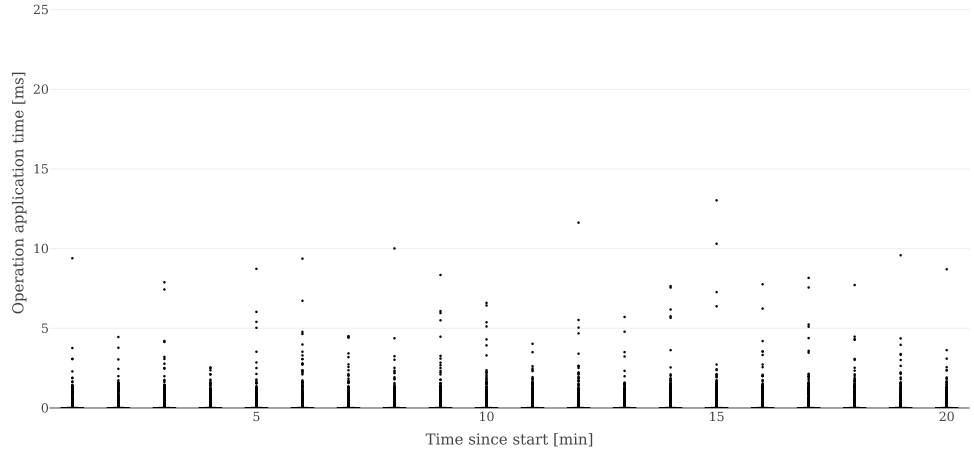

Figure 4.9 Box plot of operation application time where each operation consists of either adding 20 characters to the beginning of the document or removing 20 characters from its different parts.

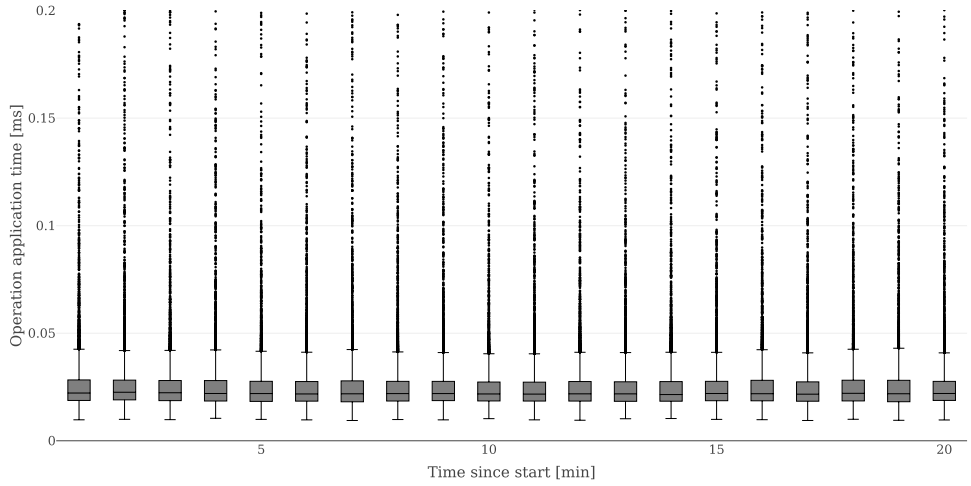

Figure 4.10 Box plot of operation application time where each operation consists of either adding 20 characters to the beginning of the document or removing 20 characters from its different parts. The y axis of the plot has a smaller range to provide more detail.

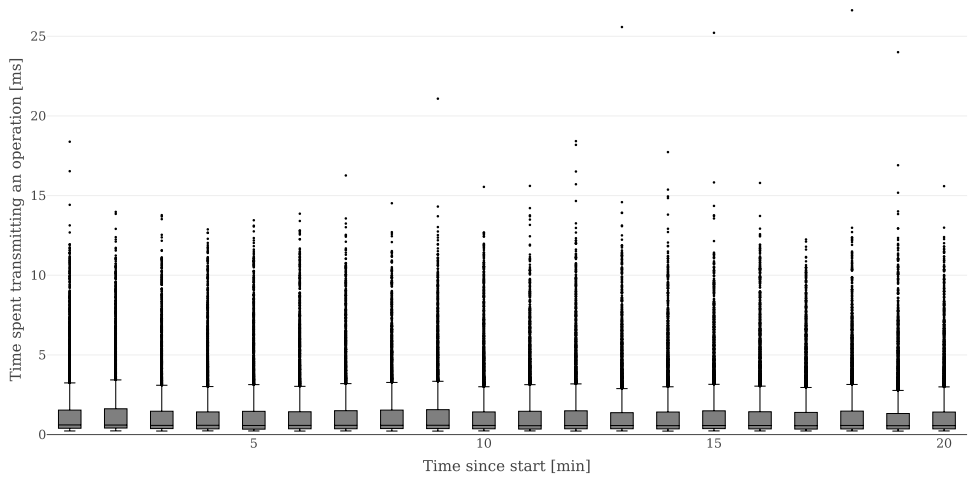

Figure 4.11 Box plot of the time it takes to transmit an operation to 200 clients.

4.3.3 Conclusion

All five listed scenarios tend to have the same performance when it comes to sending operations to clients, with the majority of sending requests being executed in less than 3 ms, with several outliers taking more than 25 ms.

Compared to the default client buffering interval of 200 ms and the network travel time that could not be measured in these scenarios, even the worst per-

forming transmission requests taking 25 ms should not be noticeable by human users.

Regarding the time it took to apply an operation to the document of the Workspace server, the measurements vary greatly based on what operations the clients send.

A general trend can be seen in the first four scenarios, where the operation application time increases over time. A likely cause is that in these scenarios the size of the document grows over time, whereas in the fifth scenario, where the operation application time seems to be constant (depicted in Figure 4.10), the document size stays bounded.

In all but the first scenario, the typing speed of users has been greatly exaggerated to see how the Workspace server performs even in extreme situations. The average user typing speed in those scenarios was 100 characters per second. However, even under these circumstances, the Workspace server seems to work efficiently, requiring less than 5 ms to apply an operation on average.

4.4 Reflexion

Encountered Problems

The biggest problem encountered during the development of this thesis was the lack of understanding of how complex operational transformation (OT) actually is. Joseph Gentle, a former Google Wave engineer, once wrote: "Unfortunately, implementing OT sucks. There's a million algorithms with different tradeoffs, mostly trapped in academic papers. The algorithms are really hard and time consuming to implement correctly, . . . Wave took 2 years to write and if we rewrote it today, it would take almost as long to write a second time."[4] This thesis also took two years to complete, and the vast majority of time was spent on operational transformation. This is also the reason why the resulting editor is just a prototype and why more features for users were not implemented.

In effect, all other problems faced were related to OT in some way or another. The major ones were how to change the system to client-server, what should the total ordering of operations be, how to make the positional system two-dimensional, what operation types should be supported by the system, among many others. The problem of how to test the system was also prominent because it required devising systems to generate failed transformations and saving all related circumstances to finally be able to reproduce them in a controlled environment.

[4]The quote can be found at the ShareJS website: https://sharejs.org/

Accomplishments

Even though the devised algorithms and the theory behind them have flaws (2.4), they should still be considered a success. The peer-to-peer model introduced in the *article* was transformed into client-server, which dramatically reduced user bandwidth requirements (1.6.1). The concept of dependence and independence was extended to handle both operations and their substituent parts (A.4.6). The introduction of operation chains (A.3) increased the intention preservation capabilities compared to the system in the *article*. Changing the positional system from one-dimensional (position) to two-dimensional (row and row position) led to faster transformations and better cooperation with the document visualization library (3.2.2), which supports the two-dimensional system.

The flaws of the algorithms can be mitigated by the resynchronization functionality, allowing users to manually synchronize.

Real Uses

The prototype itself proved to be valuable, as it was used to help solve real problems several times during the period it was developed. These problems were usually helping family members with school math problems, but the prototype was also used as a medium to easily share data with others or to hold a shared checklist of chores. Although none of these uses were related to programming, the collaborative features of the prototype still proved to be useful.

Conclusion

The main focus of this thesis was to implement a collaborative editor and repository prototype that would support many users concurrently editing documents. To do so, a new consistency theory was devised that improves the scalability of the system by changing the network architecture to client-server and utilizing the set of properties the new network architecture has.

The new theory changed many low-level aspects of how consistency was achieved when compared to the source theory while keeping the same high-level approach. A new total ordering algorithm was devised that prevents clients with higher latencies to have priority and improves the intention preservation property by introducing the concept of Operation chains, which groups client operations together. The new theory additionally allows to aggregate different user changes into a single operation and changes the positional system of operations from one-dimensional to two-dimensional. However, flaws in the new theory were identified, which require the input of users to mitigate.

The prototype that was implemented supports common operations including the creation of user workspaces, performing file operations inside them, and sharing workspaces with others using its link. A series of user roles was devised that aids user management by restricting the set of actions they can perform. Additionally, the prototype was successfully deployed and used several times in real-world scenarios.

Although the performance of the prototype was not rigorously tested, several tests were executed that imply that it does not perform badly even in extreme scenarios.

Even though the prototype lacks many user features found in modern collaborative editors, it is capable to host editing sessions of multiple clients and allows them to edit documents collaboratively.

Bibliography

[1] John Day-Richter. *What's different about the new Google Docs: Conflict resolution.* 2010. URL: https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_22.html.

[2] John Day-Richter. *What's different about the new Google Docs: Making collaboration fast.* 2010. URL: https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html.

[3] *Etherpad and EasySync Technical Manual.* Tech. rep. AppJet, Inc., with modifications by the Etherpad Foundation, 2018. URL: https://github.com/ether/etherpad-lite/raw/master/doc/easysync/easysync-full-description.pdf.

[4] C. A. Ellis and S. J. Gibbs. "Concurrency Control in Groupware Systems". In: *SIGMOD Rec.* 18.2 (1989), 399–407. ISSN: 0163-5808. DOI: 10.1145/66926.66963. URL: https://doi.org/10.1145/66926.66963.

[5] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. "Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems". In: *ACM Trans. Comput.-Hum. Interact.* 5.1 (1998), 63–108. ISSN: 1073-0516. DOI: 10.1145/274444.274447. URL: https://doi.org/10.1145/274444.274447.

[6] Rui Li and Du Li. "A New Operational Transformation Framework for Real-Time Group Editors". In: *IEEE Transactions on Parallel and Distributed Systems* 18.3 (2007), pp. 307–319. DOI: 10.1109/TPDS.2007.35.

[7] Kenneth Birman, André Schiper, and Pat Stephenson. "Lightweight Causal and Atomic Group Multicast". In: *ACM Trans. Comput. Syst.* 9.3 (1991), 272–314. ISSN: 0734-2071. DOI: 10.1145/128738.128742. URL: https://doi.org/10.1145/128738.128742.

[8] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

Appendix A

Formal Background of the Concurrency Model

This appendix holds the theoretical background of the implemented concurrency model. Concepts used in the model will be formally defined, and their interactions and implications will be explained in theorems.

High-level concepts, such as the Generic Operation Transformation Control Algorithm and the Garbage Collection Scheme were adapted from the *article*, but their inner working were almost completely remade to fit the needs of the proposed concurrency model, as the concepts proposed in the *article* could not be easily extended to enable the improvements and changes described in Section 1.6. Therefore, the following lemmas and theorems contain proofs of their correctness, as they do not have a counterpart in the *article*.

A.1 Introduction

This section will introduce the assumptions on which the theory relies, introduce definitions of basic terms used throughout this chapter, and explain the general approach of applying operations to documents.

A.1.1 Assumptions and their Implications

In order to simplify the following definitions and algorithms, a set of assumptions is considered.

Assumptions

1. Operations sent by the same client are received by the server in the same order as they were sent by that client.

2. Operations sent by the server are received by clients in the same order as they were sent.

3. All sent operations will reach their recipient.

4. All clients joined at the same time (they started from the same document state).

5. No clients join or leave.

Implications

1. If the server sends an operation to a client, by the time it arrives, the client will already have received all previous operations sent by the server.

2. All clients will receive operations in the same order.

3. The history of received operations after receiving an arbitrary operation O will be the same for all clients.

These implications are used to define *Server ordering* (SO for short) as the order in which operations are sent by the server. Due to Assumption 2., defining Server ordering as the order clients receive operations would be equivalent.

A.1.2 Server Ordering

Definition 1 (Server ordering)**.** *Given operations A and B: $A \rightarrow B$ (A was sent directly before B) if and only if:*

A was sent by the server before B and there exists no operation C that was sent after A and before B.

Definition 2 (Transitive Server ordering)**.** *Given operations A and B: $A \ ^*\!\rightarrow B$ (A was sent before B) if and only if:*

There exist $n \in \mathbb{N}_0$ operations C_i such that

$$A \rightarrow C_1 \rightarrow C_2 \rightarrow ... \rightarrow C_n \rightarrow B$$

For clients and the server to be able to determine what operation was sent before another, a structure called the *Server ordering buffer* (SOB for short) is needed to store operations as they arrive.

Definition 3 (Server ordering buffer - SOB)**.** *The Server ordering buffer is a list of operations maintained by all clients and the server independently, which contains operations in the order they are received.*

The server and each client have a local version of the SOB, which is appended whenever a new operation arrives.

For clients, this means that whenever they receive an operation from the server, it is added to their local SOB. The only means for local client SOBs to differ from each other is for some operations not arriving to some clients, but arriving to others. This will, however, be amended with time, due to assumptions 2 and 3, meaning that if all clients were to have their SOBs contain an identical number of operations, their SOBs would be equal.

This observation is used to define the *Server Ordering Index* function used to get the index of an operation inside an SOB.

Definition 4 (SOIndex function)**.** *When applied to operation O, returns its zero-based index in the local SOB. This index is equal to the number of operations sent before O.*

A.1.3 Operation Application

Received operations generally cannot be applied directly to a document. Their contexts must be determined and transformed to match that of the local document. This chapter serves as a high-level introduction to the process of operation application.

First, general problems of operation application are described. Then, definitions of essential terms needed for the description of operation application are introduced. Finally, high-level procedures for handling operation application are defined.

General Problems with Operation Application

If there was no transaction latency between clients and the server and operation application would be instant, received operations could be applied directly to the *local document*, as all documents would be equal, and all participants could immediately see the results of their actions. There would be no need to explicitly enforce document convergence because all documents would be identical due to the instant nature of the environment and the intentions of clients would be preserved automatically, as there is no latency causing other clients to possibly interfere with someone's intent.

However, Internet latency is usually high enough to be noticeable and an overwhelmed server will only add to the effect. To apply an operation to a *local document*, the document must be in the same *document state* as the one in which the operation was created. The operation needs to be applied in the correct *operation context*. Otherwise, a received operation could produce unintended

results, such as adding text to a non-existent row, deleting non-existent text, or adding text to the wrong place.

Definitions

Definition 5 (Initial document)**.** *A constant collection of text rows present at the beginning of a collaborative editing session.*

The operations which led to the creation of the initial document (for example, in a previous editing session) do not need to be considered by the algorithms described in later chapters, because it is simpler to apply new operations on an established document than to apply new operations to an empty document and a collection of operations from previous editing sessions.

The reasoning for this will be explained in later chapters and it will be exploited in the *Garbage Collection Scheme*.

Definition 6 (Current document)**.** *A collection of text rows at the present time in the premises of an entity (either client or server) formed by applying operations to the* initial document.

Definition 7 (Document state)**.** *An ordered pair consisting of the* initial document *and a list of operations that can be directly applied to the* initial document *to produce the* current document.

Definition 8 (Operation context)**.** *The context of operation O is the list of operations that defines the* document state *from which operation O was generated. The context of O does not include O.*

Definition 9 (Local document)**.** *The* document state *of a specific client or the server.*

Definition 10 (Local and external operations)**.** *Local operations are all operations that were generated by a specific client. External operations are all operations that were not generated by a specific client.*

Definition 11 (Received and unreceived operations)**.** *Received operations were received by the entity in question (client or server). Unreceived operations have not yet been received.*

Operations can be local or external with regard to a specific client. The key difference is that local operations do not need to be received by the client to be present in the local document. A new local operation is applied directly to the local document at the time of generation and is then sent to the server, which will eventually receive it and send it back to the client. The received local operation

must not be reapplied. This has the implication that newly generated operations can have yet unreceived operations in their contexts, but they must be local because external unreceived operations need to be first received in order for the client to have any knowledge of them. External operations can only be applied once they have been received by the client.

High-Level Procedures

Generally, the local document can be in a state different from that required by an incoming external operation. This conflict can be resolved in one or a combination of well-defined procedures, each of which depends on the global *total ordering*, an ordering of operations that aims to be the fairest to the intentions of all clients. Total ordering will be formally defined later, but for the purposes of this section, one can imagine the total ordering as the Server ordering, i. e. an ordering that assumes that operations received by the server sooner were generated sooner as well, and therefore should be ordered before later operations.

The high-level overview of said procedures handling conflicts between the local document state D and an incoming external operation O is as follows:

1. If D contains an operation O_D that is not in the context of O and O_D is *totally ordered* before O, O_D can be included in the context of O using a *transformation algorithm* and the transformed O can be directly applied to D.

2. If D contains an operation O_D that is not in the context of O and O_D is *totally ordered* after O, O_D can be excluded from D, O applied to the new document state and O_D reapplied (after being transformed to include O in its context).

These two procedures can be combined and generalized in order to handle more complex situations.

Situations where D does not contain an operation present in the context of O do not need to be considered because of the following theorem.

Theorem 1. *An external operation O received by a client cannot have any operation O_C in its context that would not be present in the local document state.*

Proof. In order for an operation O_C to be present in the context of another operation O, it had to be generated earlier to be present in the document state from which O was generated.

In case both O_C and O were generated by the same client, O_C had to be sent to the server first, meaning that other clients received it before O (based on Assumptions 1, 2, and 3).

In case O_C and O were generated by different clients, O_C had to be already broadcasted by the server before O was generated in order to be present in the document state from which O was generated, therefore, by the time O is received by clients, they already received O_C (based again on Assumptions 1, 2 and 3). □

A.2 Dependency

In order to understand the relationship between a document state and the context of an operation, it is necessary to understand the relationship between their individual constituent operations.

First, a more in-depth definition of operations will be introduced, and then different kinds of dependencies will be identified.

A.2.1 Operation and its Metadata

Definition 12 (Operation)**.** *An operation is an ordered pair of* operation metadata *and a* dif*.*

Definition 13 (Operation metadata)**.** *An ordered quadruplet of numbers. In order:* client ID*,* operation serial number*,* previous client ID*,* previous operation serial number*.*

Definition 14 (Client ID)**.** *A unique number that identifies the authoring client of an operation.*

Definition 15 (Operation serial number - OSN)**.** *A number that identifies the position of an operation among the positions of all other operations of the same client in time. A new operation will have its OSN one higher than the previous operation from the same client.*

Definition 16 (Previous client ID)**.** *The* client ID *of the last operation received from the server.*

Definition 17 (Previous operation serial number - Previous OSN)**.** *The* OSN *of the last operation received from the server.*

The purpose of the metadata is to determine the relative position of any operation amongst all other operations according to some ordering. The ordering used in this thesis is the *total ordering* defined later. Because Server ordering is a cornerstone of total ordering, it is necessary to link the time of generation of an operation to a specific epoch in the SO.

The operation metadata quadruplet does exactly that. The first two numbers form a unique identification for each operation. The last two numbers describe

the last operation in the SO at the time of generation, linking it to a specific time frame.

This is also why these two numbers (previous client ID and previous OSN) are not defined as the client ID and OSN of the last operation in the local document context, because such operations generally do not have a universal ordering, as they can be local and yet unreceived (thus their exact position in the SO is not yet known).

In order to work with operation metadata throughout this thesis, metadata access functions are defined.

Definition 18 (CID function)**.** *When applied to an operation, returns its client ID.*

Definition 19 (OSN function)**.** *When applied to an operation, returns its OSN.*

Definition 20 (PrevCID function)**.** *When applied to an operation, returns its previous client ID.*

Definition 21 (PrevOSN function)**.** *When applied to an operation, returns its previous OSN.*

A.2.2 Dependency Types

Definition 22 (Dependency)**.** *Operation A is said to be dependent on operation B if and only if A contains B in its context.*

Definition 23 (Direct Dependency)**.** *Operation A is said to be directly dependent on operation B if and only if:*

$$\mathrm{PrevCID}(A) = \mathrm{CID}(B) \wedge \mathrm{PrevOSN}(A) = \mathrm{OSN}(B)$$

Lemma 2. *Direct dependency is a dependency.*

Proof. Let B be the direct dependency of A. From the definition of direct dependency, it is clear that B was the last operation received before A was generated, and from the definition of operation context, it is clear that B is present in the context of A, and thus it is a dependency. □

Thus, each operation has exactly one direct dependency that can be directly identified from its metadata. It should be noted that multiple operations can have the same direct dependency, even operations made by the same client. For the latter, this can happen when multiple operations are generated in an interval when no operations are received. Such a series is called an *Operation chain* and has special implications for total ordering, described later.

If an operation was generated before any operation was received, such an operation would have no direct dependency because its previous client ID and previous OSN would be undefined. In order to enforce the invariant that each operation has a direct dependency, all SOBs at the start of the editing session can be appended by a special operation with an empty dif and a special client ID and OSN that is known amongst all participants.

Lemma 3. *For any operation A with a direct dependency B, A is dependent on all operations C that satisfy $C \ ^*\!\!\rightarrow B$.*

Proof. Because B is a direct dependency, B had to be present in the document state from which A was generated and therefore is in the context of A.

All operations C that were sent before B had to arrive before B, and therefore are also part of the document context from which A was generated, making them part of the context of A. □

Definition 24 (Local Dependency)**.** *Operation A is said to be locally dependent on operation B, given that A is* directly dependent *on C, if and only if:*

$$\mathrm{CID}(A) = \mathrm{CID}(B) \wedge \mathrm{OSN}(A) > \mathrm{OSN}(B) \wedge C \ ^*\!\!\rightarrow B$$

Local dependencies of operation A are operations generated by the same client before A which are still unreceived.

Lemma 4. *Local dependency is a dependency.*

Proof. Let L be the list of local dependencies of A. From the definition of local dependency, it is clear that locally dependent operations are a subset of all operations made by the same client that generated A, and because their OSN is lower than that of A, they were generated earlier. Because of this, they were applied to the document from which A was generated, and thus they have to be present in the context of A. □

Theorem 5. *For any operation A with a direct dependency B, A does not have dependencies other than B, local dependencies of A, and the list L of all operations C_i that satisfy $C_i \ ^*\!\!\rightarrow B$.*

Proof. The fact that B, local dependencies of A, and all operations in L are dependencies of A, was proven in the previous lemmas.

All that remains is to prove that these dependencies are the only ones that A has. First, we will prove that B and all operations in L are the only received dependencies of A. Then, we will prove that the local dependencies of A are the only unreceived dependencies of A. Because operations can be received or unreceived, the theorem is proven.

If B and all operations in L were not the only received dependencies of A, there would have to exist a received operation X that is not the direct dependency of A and X would have to be sent after B (in order to not be included in L). However, by the definition of direct dependency, B is the last received operation, and thus there cannot be such operation X.

If local dependencies of A were not the only unreceived dependencies of A, there would have to exist an unreceived operation Y that is not a local dependency of A, meaning that it was either generated by a different client than the one that generated A, had its OSN higher or equal to A, or be sent before B.

Y had to be generated by the same client that generated A, as external operations that are unreceived could not have influenced the local document in any way, and therefore Y could not be part of the document state from which A was generated.

If Y had an OSN higher than A, then Y could not be in the context of A because Y would be generated after A. $\mathrm{OSN}(Y)$ cannot be equal to $\mathrm{OSN}(A)$, as that would mean that $Y = A$ and the context of an operation does not include itself.

If Y was sent before B, it would also be received before B, which means that Y was not unreceived. □

A.3 Total Ordering

Total ordering is an ordering that applies universally to all operations in an editing session. It shall be deterministic, the same order of operations shall be obtained given a fixed set of operations with fixed metadata. It dictates in what order operations should be applied to the document.

First, the motivation behind *total ordering* will be presented. Then, the definition of *Operation chains*. Lastly, *total ordering* will be defined alongside *Chain ordering*, *Intrachain ordering*, and *Local total ordering*.

A.3.1 Motivation

A universal ordering of operations is necessary for all clients to converge to the same document state. If operations were to be applied in an unorderly fashion, some that should be applied later would be applied sooner, and vice versa, resulting in different document states across clients.

Many different orderings can be defined, but they all need to consider the three main constraints put down on them by the environment and users:

1. Universality: Enforcing the ordering shall result in the same order of operations at the premises of all participants (clients and the server). The

resulting order of operations shall be deterministic, considering that clients can gain access to operations in different orders due to local operations being applied directly, without the need to be received from the server. Given two participants with the same set of operations, the same ordering shall be derived.

2. Complexity: The more complex the ordering grows, the more complex the underlying algorithms will become. Some ordering requirements that may seem minor, yet add some value to the users, can drastically increase the complexity of the algorithms, both theoretically and computationally.

3. Usability: The ordering should aim to be as *fair* as possible. Users should not be surprised by the order of operation application. If a user has constantly higher priority than another one, the latter may grow to become frustrated that his changes are being regularly overwritten by the former. If someone's change takes way too long to be applied, it may become confusing to the author, who might begin to think that some sort of error occurred, and surprising to the other participants, as the delayed change may be dependent on a context that was in the meantime heavily modified or deleted altogether, making the change non-sensical in the current context.

The simplest universal ordering that could be applied in the client-server model proposed in this thesis would be the Server ordering. Operations would always be applied in the order the server receives them. Such an ordering assigns higher priority to users with lower latency; however, the fact that higher latency leads to worse service is usually universally accepted and thus this source of unfairness may be considered minor. The time it takes for a user's change to be applied to documents is directly proportional to latency (assuming the time it takes to transform and apply operations is not significant), meaning that no mechanism might postpone the application of a specific operation that might appear to be *unfair*. Being an ordering which forms itself automatically with the sole requirement that earlier operations have priority makes Server ordering one of the algorithmically simplest. Simpler orderings could be devised in theory if the requirement was to be ignored, but such ordering could impair general *fairness*.

However, a significant pitfall of the Server ordering was identified. When a user makes multiple changes (sends out multiple operations) from the same document context, it would be expected that unreceived operations would treat the set of sent-out operations atomically, i.e. as a single operation. It would be counterintuitive that unreceived operations would only transform part of the set, given that the whole set was generated from the same context and no operation from the set is more related to the unreceived operations than others.

An example would be two clients writing some text on the same initially empty line. Their changes would be sent out as multiple operations (so that the document state of clients is frequently updated), but due to latency, the operations from the other client do not arrive in time. The clients generated several operations that were linked to one another. Still, in case the server received the operations in an interleaved manner and total ordering was Server ordering, the operations would be applied in an interleaved manner as well, making the resulting row fragmented and hard to repair. The ideal outcome would be that all the operations from one client would be placed next to each other at the beginning of the line, and the operations from the other client would follow. This would make the text of both clients easily separable. It would also preserve their intention to write a coherent line of text, whereas the interleaved scraps of text if Server ordering was used would violate this intention.

This problem is mitigated by *operation chains*, which can be used to define an ordering that considers such sets of operations as a single operation.

A.3.2 Operation Chains

Definition 25 (Operation Chains)**.** *An* operation chain *is the longest chain of operations sent by a client during the time interval determined by two consecutive operation receivals.*

The previous definition can be interpreted as saying that, whenever a new operation is received, all operations generated locally from that time up to the time a new operation is received are composing a single operation chain.

At the start of an editing session, no operation was yet received, but again we can consider the artificial operation added at the beginning of all SOBs mentioned in the previous chapters. In this case, an operation chain starts with the start of the editing session and ends with the first operation received.

Theorem 6. *The set of all operations generated by all clients during an editing session can be aggregated into the set of all operation chains.*

Proof. Each operation was generated after an operation was received, except for the start of an editing session when no operation was yet received. In that scenario, we consider an artificial operation added to all SOBs that acts as the first received operation.

Each operation was generated before the next operation sent out by the server was received. The only special case can be the end of the editing session when no more operations are being received, but before the last operation is received, all operations already had to be generated; otherwise they would be sent out

and eventually received, but that contradicts the fact that the last operation was received.

Because all operations were generated in a time interval determined by two consecutive operations receivals, all of them are parts of operation chains. □

Theorem 7. *The operations in an operation chain have identical previous client IDs and previous OSNs. If a set of operations has the same client ID, previous client ID, and previous OSN, it is a subset of a single operation chain.*

Proof. Previous client IDs and previous OSNs identify the last operation present in the local SOB. Because during the generation of the operations composing an operation chain, no operation was received, the previous client IDs and previous OSNs have to be the same.

If a set of operations has the same client ID, they were made by the same client. To have the same previous client ID and previous OSN, all operations in this set had to be generated during a time when the same operation was the last operation present in the local SOB. This means that during the generation of all the operations in the set no operation was received; otherwise, some operations in the set would refer to that received operation with their previous client IDs and previous OSNs. From the definition of operation chains, the set has to be part of a potentially longer operation chain; therefore, the set is a subset of an operation chain. □

A.3.3 Total Ordering Definition

The total ordering that was chosen for the purposes of this thesis is the Server ordering augmented to take operation chains into account. Effectively, what an operation is to Server ordering is an operation chain to total ordering.

First, we define Chain ordering, an ordering that orders operation chains among themselves. Then we define Intrachain ordering that orders operations inside an operation chain. Finally, we use these new orderings to define total ordering.

Definition 26 (Chain ordering)**.** *Given operation chains C_1 and C_2: $C_1 \rightsquigarrow C_2$ (chain C_1 was sent directly before chain C_2) if and only if:*

There exists an operation in C_1 that was sent by the server before all operations in C_2 and there exists no operation chain C_3 such that there exists an operation in C_1 that was sent by the server before all operations in C_3 and at the same time there exists an operation in C_3 that was sent by the server before all operations in C_2.

Definition 27 (Transitive Chain ordering)**.** *Given operation chains C_0 and C_{n+1}: $C_0 \ {}^{*}\!\rightsquigarrow C_{n+1}$ (chain C_0 was sent before chain C_{n+1}) if and only if:*

There exist $n \in \mathbb{N}_0$ operation chains $C_1, C_2, ..., C_n$ such that

$$C_0 \rightsquigarrow C_1 \rightsquigarrow C_2 \rightsquigarrow ... \rightsquigarrow C_n \rightsquigarrow C_{n+1}$$

It shall be noted that for the purposes of Chain ordering, only the first-sent operations in operation chains are important. If we were to reduce all operation chains to just those operations that were sent first, the ordering of these reduced chains would be identical.

Definition 28 (Intrachain ordering)**.** *Given an operation chain C composed of the operations $O_1, O_2, ..., O_n$ and given the indices $k, l \leq n; k \neq l$: $O_k \rightharpoonup O_l$ (O_k was sent directly before O_l inside chain C) if and only if:*

O_k was sent before O_l, and there exists no operation $O_m \in C$ such that O_k was sent before O_m and O_m was sent before O_l.

Definition 29 (Transitive Interchain ordering)**.** *Given an operation chain C composed of operations $O_1, O_2, ..., O_n$ and given indices $k, l \leq n; k \neq l$: O_k $^*\!\rightharpoonup O_l$ (O_k was sent before O_l inside chain C) if and only if:*

O_k was sent before O_l.

Intrachain ordering is effectively Server ordering reduced only to the operations inside the same operation chain.

Definition 30 (Total ordering)**.** *Let $C_1, C_2, ...C_n$ be a list of all operation chains generated during the course of an editing session, ordered using the Chain ordering so that $C_i \rightsquigarrow C_{i+1}$.*

Let $O_{C_{11}}, O_{C_{12}}, ..., O_{C_{1|C_1|}}, O_{C_{21}}, ..., O_{C_{2|C_2|}}, ..., O_{C_{n|C_n|}}$ be a list of all operations generated during the course of the same editing session, where operations belonging to the same operation chain are ordered using Intrachain ordering so that $O_{C_{ij}} \rightharpoonup O_{C_{ij+1}}$.

The ordering of such a list is the total ordering.

Total ordering, therefore, orders operations using both Chain ordering and Intrachain ordering. However, due to the nature of it requiring all operations generated during an editing session in order to order them, it is impractical to use total ordering directly when determining the order of operation application, as clients generally do not have access to all operations generated. For that reason, the Local total ordering is defined.

Definition 31 (Local total ordering)**.** *Let T be the list of all operations generated during an editing session ordered using total ordering. Let R be the set of all received operations of the local client. Let L be a list that is the intersection of T and R, where the order of operations from T is preserved. The ordering of list L is the Local total ordering.*

Definition 32 (Consecutive Local total ordering)**.** *Let L be a list composed from the set of all received operations by the local client, ordered using Local total ordering, and let A and B be operations from L. $A \Rightarrow B$ (A is locally ordered directly before B) if and only if:*

The index of A in L is exactly one less than the index of B in L.

Definition 33 (Transitive Local total ordering)**.** *Let L be a list composed from the set of all received operations by the local client, ordered using Local total ordering, and let A and B be operations from L. $A \ ^*\!\Rightarrow B$ (A is locally ordered before B) if and only if:*

The index of A in L is less than the index of B in L.

Local total ordering is total ordering defined on the set of all received operations instead of all generated operations.

A.4 Transformation Algorithms

In order to preserve the intentions of clients, operations need to be transformed to reflect the changes made by all operations, that were totally ordered before it.

The *Generic Operation Transformation Control Algorithm* (GOTCA), *List Include Transformation* (LIT) and *List Exclude Transformation* (LET) as well as the generic scheme of Primitive Include and Exclude Transformations (IT, resp. ET), defined in the *article*, were kept and adapted to the changes in this thesis.

The main differences are the way total ordering is ascertained, the introduction of Local and Direct dependency, Operation chains, and that operations can now include multiple dif elements.

The fact that operations can include multiple dif elements led to the creation of new algorithms needed in the overall process of transformation, those being *MakeDependent* and *MakeIndependent*, that change the relationships of the dif elements contained in the dif of an operation.

GOTCA and UDR had also been redefined to work with difs rather than dif elements.

First, the data structures used by the algorithms will be defined alongside the identified special transformation results. Then, the transformation algorithms will be defined, starting with those of the lowest level.

A.4.1 Dif Elements

Dif elements take on the role of data structures, which were called *operations* in the *article*. They are the smallest units of change, characterized by a simple effect, that will be applied to a specific position within the document.

Dif Element Types

The *article* defined two dif element types (or operations), *Insert* and *Delete*, that inserted a string of text, resp. deleted a set amount of characters, at a certain position.

This thesis expands the set of types with two new entries, *Newline* and *Remline*, that handle the addition, resp. removal of row separators (newline characters). The reasoning for this addition was already introduced in Chapter *1.5.2*. Additionally, the Insert and Delete types were renamed to *Add* and *Del*, respectively, and can no longer add or remove row separators.

Dif Element Definitions

All dif element types have two positional parameters, called *Row* and *Position*, represented by integers, describing the row number and the row character position of the dif element, respectively. Additionally, the Add type contains a parameter containing the added text called Content, and the Del type contains the count of deleted characters called Count. The Newline and Remline types do not contain any additional parameters.

Definition 34 (Add Dif Element Type)**.** *The Add dif element type is an ordered triplet consisting of Row, Position, and Content. When applied to a document, the text contained in Content is inserted after the character at the specified Row and Position, possibly moving some text to the right. Content must not contain characters that can be interpreted as row separators.*

Definition 35 (Del Dif Element Type)**.** *The Del dif element type is an ordered triplet consisting of Row, Position, and Count. When applied to a document, a continuous text segment starting with the character specified by Row and Position and of the length equal to Count is removed for it, possibly moving some text to the left. A Del must not delete a row separator in the context where it was generated.*

Definition 36 (Newline Dif Element Type)**.** *The Newline dif element type is an ordered pair consisting of Row, and Position. When applied to a document, a line separator is inserted after the character at the specified Row and Position, creating a new and empty row to which all the text following the character specified by Row and Position is moved. The moved text had to be on the same row as the specified character. The row numbers of rows, that were originally positioned at row number* $Row + 1$ *or higher, are incremented by one.*

Definition 37 (Remline Dif Element Type)**.** *The Remline dif element type is an ordered pair consisting of Row, and Position. When applied to a document, the line separator situated directly after the character specified by Row and Position*

is removed, appending the row with the text of the following one. The parameters of a Remline always target the last position of a row in the context where it was generated. The row numbers of rows, that were originally positioned at row number $Row + 1$ *or higher, are decremented by one.*

A.4.2 Lost Information

During transformation, it is possible for a dif element to lose information. Transformed dif elements that lost information cannot be returned to their original form by executing an inverse transformation on them without some additional information being stored beforehand. The handling of lost information is done in the same way as proposed in the *article*; if a transformation happens to be lossful, the version of the dif element before executing that specific transformation is saved, and a link to the other dif element participating in the transformation, the transformer, is made. If the dif element were to be transformed with the inverse transformation and the same transformer, the saved, pre-transformation form of it would be used to get the correct transformation result.

A.4.3 Relative Addressing

When a dif element A is positioned inside the effect range of another dif element B (i.e., B added some text and A made some changes to it), that is totally ordered before A, excluding B from A makes the position of A undefined. A was in a sense anchored to B, and the removal of B results in there not being a correct placement for A. This situation is handled by making A relatively addressed to B during the exclusion, making the main positional reference point of A not the start of the document but the start of B. After B is included back into the context of A, it will be possible to calculate its absolute position based on the relative position of A and the absolute position of B. During subsequent transformations after the exclusion, the position of A would not be changed by any transformation as long as its position is relative. Relative addressing occurs only during exclusion transformations, which are bound to be reversed in the future with an include transformation before the transformation result is applied to the document, meaning that no mechanism of how to apply relatively addressed dif elements to a document has to exist.

As proposed in the *article*, relative addressing can only occur when the excluded transformer is an Insert operation. In this thesis, due to Insert being split into Add and Newline, relative addressing can happen when excluding any of these two dif element types. The Newline behaves as a single character for the purposes of relative addressing, meaning that only dif elements that are positioned at the same position as the Newline will become relative.

A.4.4 Fragmentation

As identified in the *article*, transforming a dif element may lead to its fragmentation. Originally, this could happen in the following situations:

1. Including an Insert operation into the deletion range of a Delete operation, which makes the Delete remove a segment before and after the Insert, as it must not change the content of the Insert.

2. Excluding a Delete operation from the deletion range of another Delete, effectively replacing the deletion range of the excluded Delete with text. Similarly to the first scenario, the Delete will be split into two segments deleting characters before and after the added text.

3. Excluding an Insert operation from a Delete that party removed its text and then some more. The Delete will fragment into two, the part removing the text of the excluded Insert, which will become relatively addressed to it, and the second part, which removed the text not inserted by the excluded Insert. The reason for this split is to enforce that an operation can be positioned either absolutely or relatively, not a combination of both.

With the addition of new dif element types, dif element fragmentation can also occur in the following transformations:

1. Including a Newline into a Del; effectively the same as including an Insert into a Delete.

2. Excluding a Remline from a Del; effectively the same as excluding a Delete from a Delete.

One can notice that excluding a Newline from a Del is not on this list. While excluding an Insert from a Delete could lead to fragmentation, excluding a Newline from a Del does not, as a Del cannot remove a range containing a line separator, thus the line separator could not be in the deletion range of the Del.

Newlines and Remlines cannot be fragmented, as they effectively add, resp. remove a single character.

A.4.5 Wraps

To store the additional information when handling lost information and relative addressing, a data structure called a *Wrap* is introduced.

Definition 38 (Wrap)**.** *A data structure that wraps a dif element with the data necessary for its transformation. A wrap contains the current form of a dif element, a unique identifier, whether it lost information or is relatively addressed, and if so, the original form of the dif element before it lost information, the transformer that caused the loss, and the transformer to which it is relatively addressed.*

Primitive transformation algorithms usually only work with wraps so that they can directly access the additional data it holds.

A.4.6 Difs, Wrapped Difs, and Dif Dependency

Definition 39 (Dif)**.** *A list of dif elements.*

Definition 40 (Wrapped Dif)**.** *A list of wraps.*

Operations sent by clients can contain multiple dif elements stored in a dif. These difs are special, as all dif elements they contain are dependent on all previous dif elements in that dif. This is because the dif elements were generated one after another, making the latter dependent on the previous ones. Such a dependent sequence of dif elements acts atomically in a sense; the operation metadata used to determine the total ordering is shared for all dif elements of a dif, making them inseparable.

Because total ordering determines the ordering of operations, and thus difs, many transformation algorithms were redefined to work with difs instead of dif elements, as the Local total ordering they usually need to determine can be determined only once for the whole dif, optimizing the overall transformation.

Dif Dependency

Although the term *dependency* is usually reserved for operations in this thesis, the same types of dependencies can be defined on the dif element level, as was the case in the *article*. Luckily, this is not necessary here, as the high-level transformation algorithms that work with dependencies only handle transformations of whole operations, never of individual dif elements. However, the LIT and LET transformation algorithms require to distinguish between dependent and independent dif elements inside a dif, due to their inner workings, and therefore a dif-only scoped dependency is introduced.

Definition 41 (Dif Dependency)**.** *A dif is said to be dependent if and only if its constituent dif elements form an uninterrupted subsequence of the sequence present in any operation dif, or in the joined difs of an uninterrupted sequence of operations, where the context of operation O_2 is the context of the previous operation O_1, appended by O_1.*

A dif is said to be independent if and only if its constituent dif elements are the first dif elements present in operation difs of operations with the same context.

The definition considers operation difs as the source of dependent difs. If an uninterrupted subsequence of operations were to be extracted from any HB and their difs merged into one, it would also be dependent.

Independent difs are not naturally encountered; they have to be made from dependent difs using the *MakeIndependent* algorithm.

Making Difs Dependent and Independent

The high-level algorithms often utilize the *MakeDependent* and *MakeIndependent* algorithms, as the *LIT* and *LET* list transformation algorithms require their input dif to be independent, and produce, in case of *LIT* dependent, in case of *LET* independent difs. Because the difs encountered in the History buffer (a data structure defined in the next section) operations have to be dependent, as well as the difs in operations sent by clients, the high-level transformation algorithms need to make sure that independent difs are not produced.

In order to make a dif dependent and comply with the definition of dif dependency, an operation would have to be constructed from the constituent independent dif elements. Operations are constructed by capturing the changes a user makes to the local document, converting these changes to dif elements, and aggregating them into a dif. The *MakeDependent* algorithm simulates the construction of an operation, given a series of independent dif elements, and returns a dependent dif.

During operation construction, the changes made by the user depend on each other. Writing at the start of a row and then at the end makes the position of the dif element representing the latter change shift to the right; the dif elements are dependent naturally. When creating dependent difs from independent dif elements, the dif elements need to be transformed to reflect the effective changes the previous dif elements would have made to a hypothetical document. Thus, when the *MakeDependent* algorithm transforms a dif element at a certain position in the independent dif, it includes all previous dif elements so that the transformed one reflects all of their effects.

The *MakeIndependent* algorithm is the opposite. When it transforms a dif element at a certain position in a dependent dif, it excludes all previous dif elements.

A.4.7 History Buffer

The *History buffer* (HB) is a data structure that closely resembles in functionality the data structure with the same name defined in the *article*. It stores received operations, similarly to the *Server ordering buffer* (SOB). Unlike the SOB, which is ordered using the Server ordering, the HB is ordered using the Local total ordering. In addition, all operations contained in the HB were transformed to be dependent on all operations locally ordered before it (as defined by Local total ordering) and none other. A high-level overview describing how the HB is being used follows:

1. An external operation O is received by the client.
2. The UDR scheme is used to undo operations stored in the HB that are not locally ordered before O, removing the undone operations from the HB and undoing their effects from the local document.
3. O is transformed using the GOTCA to fit the local document state, applied to the local document, and pushed to the HB.
4. O is added to the contexts of the undone operations, which are then reapplied to the local document and pushed to the HB.

The HB, as its name suggests, holds the history of the local document. It shall be noted, however, that the history is not chronological (i.e. not ordered using Server ordering) but is instead ordered using Local total ordering.

The operations stored in the HB can be undone one by one in reverse order (from latest to oldest) without the need for any transformations. If all operations were to be undone in this way, the resulting document would be identical to the initial document. The undone operations could then be reapplied to restore the document to its previous form.

A.4.8 Primitive Transformations

Primitive transformations are the simplest transformations used. The goal of each primitive transformation is to transform an input dif element against another dif element, a transformer. While the input dif element is changed in the process, the transformer always stays the same. Usually, dif elements are used in their wrapped forms, as the additional data stored in wraps may have to be considered. A single high-level transformation usually performs many primitive transformations in order to reach its goal. Due to their nature, most of the theory in this thesis does not apply to them directly, as primitive transformations do not consider

things like dependencies, orderings, or data structures such as the HB, SOB, and operation metadata.

Primitive transformations are grouped into two categories: include, and exclude. There is a primitive include and exclude transformation for each ordered pair of dif element types, resulting in a grand total of 32 primitive transformations in the context of this thesis. Due to this, the pseudocode of each one cannot be found in this thesis; however, it is provided for a single primitive include and exclude transformation for illustration. The curious reader can examine the attached source code, which contains an implementation of all primitive transformations in both *C#* and *JavaScript*.

In the *article*, primitive transformations add or remove the transformer to, resp. from the end of the context of the input dif element. This thesis, however, defines context only for operations and does not recognize it for dif elements. As the sole purpose of context is to determine how an operation will be transformed, it is not necessary to introduce an analogy for it to dif elements, as the specific transformation approach is already determined by the high-level transformation algorithms. If such an analogy were required, the simplest way would be to imagine that all operations only contain difs with a single dif element; clients would split their generated operations before sending them to the server if necessary. Such operations would still have all characteristics of general operations, as the algorithms defined to work with them do not require a specific number of dif elements. The context of the operation could then be considered to be the context of its dif element, in a sense simulating the approach in the *article*.

Primitive Include Transformations

Primitive include transformations aim to include the effect of the transformer to the input dif element. The inclusion is done by simulating the application of the transformer to a hypothetical document containing the input dif element, changing its data to reflect its new position within the document.

If the transformer was an Add, had the same Row as the input dif element, and was positioned before it, the resulting effect would be that the input dif element would be moved to the right (its position would be increased by the number of characters in the transformer).

Primitive Exclude Transformations

Primitive exclude transformations are used to undo a previously applied primitive include transformation; therefore, the process has to have the opposite effect of the corresponding primitive include transformation. Ideally, given an input dif element I and a transformer T, including and then excluding T to, resp. from

I should again produce I, making primitive exclude transformations the inverse of the corresponding include transformations. However, due to the fact that primitive transformations always take a pair of dif elements as inputs and some can cause dif element fragmentation, not all transformations have a corresponding inverse transformation. These scenarios are handled by using multiple primitive exclude transformations that jointly act as the inverse.

A.4.9 List Transformations

Similarly, as primitive transformations transform a dif element against another, list transformations transform whole difs. The *article* defined such transformations as well, called *LIT* (list include transformation) and *LET* (list exclude transformation), but these served a different purpose. The idea was to transform a single dif element against a list of dif elements, and the situation when dif elements fragmented were treated as special cases. The list transformations in this thesis expand on the idea, allowing to transform any dif against another.

Dif Transformation Dependencies

If the new versions of LIT and LET were to be implemented simply by calling the *article* versions on each dif element in the input dif, the result would be corrupted. This is because the effects consecutive dif elements in a dependent dif have on each other are the last ones influencing their form. If dif element context was to be defined, the i-th dif element in a dependent dif would have the previous $i - 1$ dif elements as the last entries in its context. By using the *article* LIT on each dif element of a dependent dif, the transformed dif elements would not have this property, as the last entries of their dif element contexts would be the dif elements of the transformer. Similarly, using the *article* LET on a dif element of a dependent dif would also produce the incorrect result, as a dif element can only be excluded if it is the last entry in a dif element context (as described in the *article*).

Making the input dif independent before using list transformations on it would allow the use of the *article* LIT and LET on each dif element. The new versions of LIT and LET do not do it this way for performance reasons, but the effect is the same. The optimizations in the new versions lead to the resulting dif of LIT being dependent and the result of LET being independent by default.

A.4.10 Generic Operation Transformation Control Algorithm

The purpose of GOTCA is to transform the context of an operation to fit the local document state. That is, the new context must contain all operations locally ordered before the transformed operation and none other. This section will describe how this version of the GOTCA differs from the version in the *article*, provide pseudocode derived from the *article* pseudocode, and then explain the individual steps.

Changes in GOTCA

The main functional difference from the GOTCA defined in the *article* is that this version can process the whole operation dif instead of a single dif element. This change was made so that the high-level process of handling operations remains as close to the way it was done in the *article* as possible, that is, the received operation is processed using the *Undo/Do/Redo* (UDR) scheme, which potentially uses the GOTCA.

This version can handle whole operation difs because the dif elements share the same operation metadata. Due to this, determining dependencies can be done just once, and besides the *MakeIndependent* and *MakeDependent* algorithms used, the resulting transformation process closely resembles the one proposed in the *article*.

Finding Operation Dependencies

To change the context of operation O to fit the document state, it is necessary to know which operations should be added to it and which should be removed. Let C_O be the context of O, HB the History buffer of the document, and SOB the Server ordering buffer. The resulting context of the transformed O, denoted $C_{O'}$, shall contain all operations in HB that are locally ordered before O. According to *Theorem 1*, no operations need to be removed from C_O, as $C_O \subseteq$ HB.

Because operations do not contain a data structure that contains their context, it needs to be found manually. *Theorem 5* lists all dependencies of O, and thus all operations contained in C_O, and can serve as a guide for finding them. Given the SOB, direct and local dependencies can be identified directly from their definitions (*Definition 23*, *Definition 24*), and the remaining dependencies can be identified using *Lemma 3*.

As $C_{O'}$ has to contain all operations locally ordered before O, the corresponding operations in the HB have to be found, as the operations in the SOB are ordered using Server ordering. Finding the operations in the HB can be done by

Algorithm 1 The Generic Operation Transformation Control Algorithm

1: $\text{HB}' \leftarrow$ Operations in HB that are locally ordered before O
2: $O_k \leftarrow$ First operation in HB' that is not a dependency of O ▷ Operation indices are the indices they had in HB
3: **if** $O_k = \emptyset$ **then return** O
4: **end if**
5: $\text{HB}'' \leftarrow$ Operations in HB' that are locally ordered after O_{k-1}
6: $DL \leftarrow [O_{i_1}, O_{i_2}, ..., O_{i_m}]$, where O_{i_j} is the j-th operation in HB'' that is a dependency of O
7: **if** DL is empty **then return** $\mathit{LIT}(O, \text{HB}'')$
8: **end if**
9: **for all** $O_{i_j} \in DL$ **do**
10: $\quad ExO_{i_j} \leftarrow \mathit{LET}(\mathit{MakeIndependent}(O_{i_j}), \text{HB}[k, i_j - 1]^{-1}$
11: **end for**
12: $InDL \leftarrow []$
13: **for** $j \leftarrow 1, m$ **do**
14: $\quad InDL \leftarrow InDL + \mathit{LIT}(ExO_{i_j}, InDL)$
15: **end for**
16: $ExO \leftarrow \mathit{LET}(\mathit{MakeIndependent}(O), InDL)$
17: $InO \leftarrow \mathit{LIT}(ExO, \text{HB}'')$ **return** InO

simply looking up operations with the same operation metadata as that of the dependencies identified in the SOB.

Now, that the dependencies of O can be identified in the HB, the first 8 steps of the GOTCA pseudocode can be executed.

Restoring the Original Forms of Dependencies

The next step of GOTCA is to restore the identified dependencies to the form they had when O was generated. Let O_k be the first independent operation locally ordered before O. All dependencies of O that are locally ordered before O_k have to already be in their original forms. If they were not, they would have to contain an operation O_m in their context that would be an independent operation of O. That would, however, make O_k not the first independent operation locally ordered before O.

It is thus sufficient to restore the state of dependencies locally ordered after O_k. Let DL be the list of such dependencies in the form they have in the HB. To restore them to their previous forms, all independent operations of O need to be excluded if they are locally ordered before them.

First, for each operation O_{ij} in DL, the sequence of operations starting with O_k up to the operation locally ordered directly before O_{ij} is excluded from it. Let these excluded forms be denoted as ExO_{ij}.

These excluded forms share the same context, and this context is the intersection of the contexts of all operations present in DL and O.

To restore the operation ExO_{ij} to the form it had when O was generated, all of the operations ExO_{il}, where $l < j$, need to be included to it. This is done by introducing the list $InDL$ which holds the restored forms, which is iteratively appended with newly restored operations. Restoring operation ExO_{ij} can be thus done by including $InDL$ to it using LIT, where $InDL$ contains the restored forms of ExO_{i1} up to ExO_{ij-1}.

Transforming the Received Operation

Once $InDL$ contains the dependencies of O in their original forms, they can be excluded from O using LET, producing its excluded form denoted as ExO. The context of ExO is the sequence of the HB starting from the beginning up to the one that is locally ordered directly before O_k. This means that the remainder of the HB can be directly included using LIT, producing the final form InO.

Appendix B

Deployment

This chapter details how the prototype should be deployed.

The source code of the prototype, named *ShEd* (Shared Editor), can be found on GitHub[1] under the MIT license. This chapter is mostly identical to the *README* file present in the GitHub repository.

B.1 Overview

ShEd is composed of two servers that need to be running to provide full functionality. The first one is the *controller server*, which serves static pages. The second one is the *workspace server*, which handles the collaborative aspects.

Users do not need to interact with the endpoints of the workspace server; this is handled by the client provided by the controller server.

B.2 Installing ShEd

To install ShEd, first, clone the repository by using *git clone*.

Then, initialize the Ace Editor[2] submodule by running the following commands in the root of the repository.

1. git init
2. git submodule init
3. git submodule update

ShEd can be started either manually, or by using Docker. If you prefer to run ShEd in Docker, you can ignore the following installation steps.

[1] ShEd repository https://github.com/eceltov/shed

[2] Ace Editor repository https://github.com/ajaxorg/ace-builds

B.2.1 Manual Installation

In order to start ShEd manually, you will need Node.js and .NET 6 SDK installed.

To install all necessary Node packages, execute:

- npm install

The final step is to bundle the client. This can be done by running the following command:

1. npm run build-client

B.3 Running ShEd

B.3.1 Running ShEd in Docker

In order to start ShEd in the Docker environment, make sure that you have Docker installed.

All components of ShEd can be started using a single command:

- docker compose up

In case you do not have the required base images downloaded, you can download them using:

1. docker pull node:18.16.1-slim
2. docker pull mcr.microsoft.com/dotnet/runtime:6.0

Please note that the containers print information about which port they use inside the container, not on the host environment. To check which ports the containers expose, open the .env file in the root of the repository. By default, the controller server uses port 8060 and the workspace server uses port 8061.

B.3.2 Running ShEd Manually

To run ShEd manually, you first need to do all the manual installation steps. Once installed, you can start the controller server by running:

- npm run start-controller-server

And to start the workspace server, run:

- npm run start-workspace-server

B.4 Using ShEd

ShEd can be accessed through the controller server at `http://localhost:8060` by default.

Once you load the page, you will be prompted to log in. By default, two demo accounts are available. Their credentials are *demo*, *password* and *demo2*, *password*, respectively.

Once logged in, you will see a list of available workspaces. You can access an existing one by clicking on its name, or create one by clicking on the *New* button. Both default users will have access to the *Demo Workspace*.

User *demo* has all privileges inside the workspace, but user *demo2* has only the rights to view and edit documents. Many of the features below are thus not available for user *demo2*.

Inside a workspace, you can do basic file operations using the buttons in the top left and open documents and folders by navigating through the file system. Opening a document will create a new tab and display the contents of the file.

To edit a document and see the effects from the point of view of a different client, you can simply duplicate the window, because a single user can have multiple clients. To see a workspace from the perspective of multiple users at once, you can open the workspace in a different browser that does not share cookies with your primary one. Alternatively, you can also open the workspace in incognito mode, because it usually does not share cookies with the main browser.

To change the role of a user in the workspace, navigate to the *Options* in the top right corner of the workspace, type in their username, select their new role, and finally change it using the *Change Role* button. The same interface can be used to add new users to the workspace or to delete the workspace.

The workspace can be in one of three access modes:

- Privileged: allows only authenticated users with access to the workspace to connect to it.
- Everyone with link: additionally allows unauthenticated users to connect to the workspace and edit documents.
- Everyone with link (read-only): does not allow unauthenticated users to edit documents.

Authenticated users can still make changes to the workspace if their role is sufficient in the latter two modes.

B.4.1 Managing Users

Adding users, removing them, and changing their passwords can be done using three utility scripts. The following example adds a user *test* with password *password*, changes its password to *new*, and finally deletes it.

1. node dev/userOperations/createUser.js test password
2. node dev/userOperations/changeUserPassword.js test new
3. node dev/userOperations/removeUser.js test

To use the scripts, it is necessary to have Node installed and the packages initialized by running:

- npm install

B.5 Configuring ShEd

ShEd can be configured to run on different ports and to use a different secret for JWT tokens. Additionally, whether it is possible to enable debug logs and the buffering window for Workspace clients.

The servers do not support hot reloading; to see the effects of the new configuration, you need to restart them.

There are two files used for configuration. The first one is the *.env* file in the root of the repository. The second one is the *config.json* file located in *dev/volumes/Configuration/*.

When running manually, the *.env* file does not need to be changed, as it will have no effect. Everything can be changed in the *config.json* file.

- To change the JWT secret, navigate to the *JWT* section and modify its *Secret* field.
- To change the ports and the endpoint on which the servers run, go to the *FallbackSettings* section and modify the *controllerServerPort*, *workspaceServerPort*, and *workspaceServerUrl*.
- To see debug logs, change *ShowDebugLogs* to *true*.
- To change the client buffering interval, navigate to the *Client* section and modify the *bufferingIntervalMilliseconds* field.

When running in Docker, the *config.json* file should be used only when configuring options that are not present in the *.env* file.