NoteSpace

Multi-platform Web Application for Real-Time Document Collaboration & Sharing

Guilherme Ferreira 49472
Ricardo Costa 49511

Supervisor: Paulo Pereira, ISEL

Beta version for report written for Project and Seminar BSc in Computer Science and Computer Engineering

June 2024

Instituto Superior de Engenharia de Lisboa

NoteSpace

Multi-platform Web Application
for Real-Time Document Collaboration & Sharing

49472 Guilherme Filipe Cardoso Ferreira

49511 Ricardo Manuel Sousa Costa

Supervisor: Paulo Pereira

Beta version for report written for Project and Seminar BSc in Computer Science and Computer Engineering

June 2024

Abstract

Both modern real-time collaboration apps like Google Docs and version control platforms like GitHub have become the standards when it comes to collaborative work. Nevertheless, both of them lack major features the other approach addresses; while both of them are sufficient for either basic real-time collaboration or fully asynchronous collaboration, it creates a major gap between them. What if there was an hybrid approach? This kind of application would allow out of the box real-time collaboration, while still taking advantages of features from version control platforms. We try to address that question through our project, NoteSpace. NoteSpace is a multi-platform web application that provides markdown-supported document based repositories, real-time collaboration, offline editing, version control and a public sharing platform.

Keywords: real-time collaboration, version control, multi-platform web application, markdown, offline editing, public sharing platform.

Resumo

Tanto as aplicações modernas de colaboração em tempo real (ex. Google Docs) como as plataformas de sistemas de controlo de versões (ex. GitHub) tornaram-se essenciais no trabalho colaborativo. No entanto, existem algumas lacunas presentes numa abordagem que são encontradas na outra. Isso levanta a questão: e se existisse uma abordagem híbrida? Esse tipo de aplicação permitiria colaboração em tempo real sem qualquer dependência externa, mas tiraria também partido das vantagens de plataformas de controlo de versões. Tentamos abordar essa questão através do nosso projeto, NoteSpace. NoteSpace é uma aplicação web multiplataforma que fornece repositórios de documentos com suporte markdown, colaboração em tempo real, edição offline, controlo de versões e uma plataforma de partilha de pública.

Palavras-chave: colaboração em tempo real, controlo de versões, aplicação web multiplataforma, markdown, edição offline, plataforma de partilha pública.

Contents

List of Figures

List of Acronyms

API Application Programming Interface

BFS Breadth First Search

CI/CD Continuous Integration and Continuous Delivery/Deployment

CLI Command Line Interface

CRDT Conflict-Free Replicated Data Type

CSS Cascading Style Sheets

DFS Depth First Search

DOM Document Object Model

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JSON JavaScript Object Notation

NPM Node Package Manager

OT Operational Transformation

REST REpresentational State Transfer

SCSS Sassy Cascading Style Sheets

SPA Single Page Application

SQL Structured Query Language

UI User Interface

VCS Version Control System

Chapter 1

Introduction

For a long time, project collaboration often relied on the exchange of local copies on each version. However, with the emergence of Git in 2005, a distributed version control system, a significant shift was made, by enabling asynchronous collaborative work while tracking each user's changes. Through it, collaborators could keep a local copy of the repository, which was a contrasting difference with previous centralized repositories, where a single instance was stored in companies' servers and all collaborations would have to be made locally. Now, changes can be made from anywhere and pushed to the centralized repository, allowing for greater flexibility.

A year later, Google Docs emerged with its groundbreaking approach to collaboration, addressing the challenges posed by traditional project collaboration methods and even Git. With its real-time collaboration features, multiple users could now edit the same document simultaneously from anywhere, across multiple devices. This meant no local copy was even needed and users could still enjoy the freedom and flexibility given by previous tools.

However, many problems still exist within both approaches: version control system (VCS) solutions like Git and GitHub lack the aforementioned real-time collaboration features, as users need to rely on other platforms to enable real-time collaboration. This requires that at least one user keeps a copy of the repository's content, which can be undesirable. Moreover, most platforms don't have mobile support. Similarly, real-time collaboration applications like Google Docs are missing some major features like an explicit VCS, as the granularity of the content of each saved version is ambiguous and dependent on the system's implementation; it also lacks a public platform for sharing documents and repositories, which is only done though link sharing.

Most tools already address most problems that emerge with collaboration, such as version merging, version control and real-time collaboration; what seems to be lacking is an approach that bridges both solutions, providing the same robust features as well as the freedom and control needed for efficient organization, sharing and collaboration on repositories.

What we aim to achieve is a combination between version controlled repository management and real-time collaboration, with emphasis on user experience. The main goal of this project is to develop a collaborative markdown-supported text editor with offline editing, workspace management, a document sharing platform and a simple version control system. Ultimately, this project seeks to create a comprehensive tool that supports efficient and productive collaboration for teams.

1.1 Outline

This document is composed of seven chapters and aims to provide a comprehensive understanding of the project. Chapter 2 will present the underlying background of collaborative editing problems and solutions. Chapter 3 will describe the problem, the challenges that arise and the solution's requirements. Chapter 4 will present the project's architecture, as well as the different modules and technologies that comprise it. Chapter 5 delves into the methodology used to implement the different features, some theoretical details, how they are structured and how they communicate. Chapter 6 explains how the project was tested. Finally, chapter 7 wraps the report addressing what was achieved and what still needs to be done.

Chapter 2

Background

2.1 Collaborative Editing

Collaborative editing enables multiple users to edit shared resources simultaneously. This process has gained a significant importance with the increase in remote work and teamwork. Collaborative editing system can be divided into two main approaches: asynchronous and synchronous editing, each with its distinct methodologies and conflict resolution strategies. For simplicity, we will address this by focusing on document editing.

2.1.1 Approaches

Asynchronous Collaborative Editing

Asynchronous collaborative editing allows multiple users to make changes to a local copy of a document, independently. This approach requires manual commits, where each user's changes need to be merged into the main version of the document at a later time. This technique is used in version control systems, such as Git, where changes are committed and then merged, potentially leading to conflicts that need to be later manually resolved.[2]

Synchronous Collaborative Editing

Synchronous collaborative editing, on the other hand, involves automatic synchronization of changes in real-time, where users can see each other's changes instantaneously. This approach eliminates the need for manual commits but introduces the challenge to manage concurrent changes automatically.

A common method for handling concurrency in general is pessimistic locking. In this case, that would mean a user would need to lock a document or a portion of it while making changes. This method is, however, not practical for collaborative editing since it prevents multiple users to edit the same document simultaneously. Hence, a strategy with a optimistic locking approach is required, which would need to allow multiple users to edit concurrently while still resolving potential conflicts.

2.1.2 Conflict Resolution Algorithms

The two main properties that should be guaranteed in a conflict resolution algorithm, is commutativity and idempotency. Commutativity ensures that the order in which operations are received and applied does not matter. Idempotency guarantees that the number of times an operation is applied does not affect the outcome.

In this example, a user performs an insertion and the other user perform a deletion in the same position. After both users apply both operations, they reach an inconsistent state, with each one observing a different state of the document, meaning that the operations do not commute.

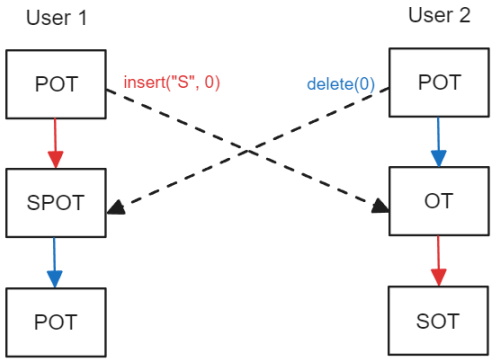

Figure 2.1: Lack of commutativity of operations, image based on the Conclave case study [1]

In this other example, both users perform the same operation, by deleting the first character in the document at the same time. In this case, both of them want to delete the same character. In this case, both states converged, but the same operation was applied more than once, meaning that the operations are not idempotent.

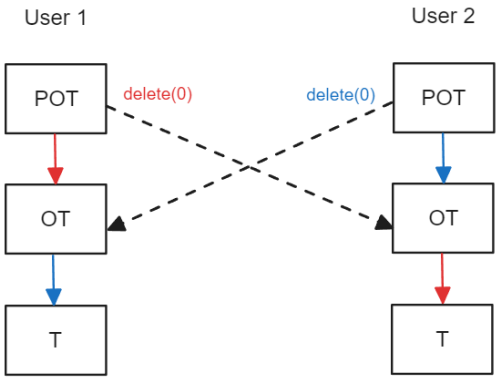

Figure 2.2: Lack of idempotency of operations, image based on the Conclave case study [1]

In order to ensure state consistency across all users editing a resource, we need an conflict resolution algorithm. This algorithm effectively guarantees that the same state should always be eventually reached across all clients, regardless of the type of operations performed by each. There are two main types of conflict resolution algorithms: Operational Transformation (OT) and Conflict-Free Replicated Data Types (CRDTs).

Operational Transformation (OT)

In OT algorithms, all users editing the same document need to communicate through the same server instance. All user originated operations in a given time frame are then processed by the server and transformed independently per user, in order to reach a state consistency, even though the operations received by each client can diverge; however, this approach comes with some drawbacks, mainly regarding scalability.

The core mechanism of the OT algorithm relies on a transformation function that modifies operations based on the context of the preceding operations before being applied. For example, if in a document, a user inserts a character and another user deletes a character at the same position, these operations might conflict. This function modifies the operations ensuring both users end up with the same desired document state, as seen in the figure 2.3.

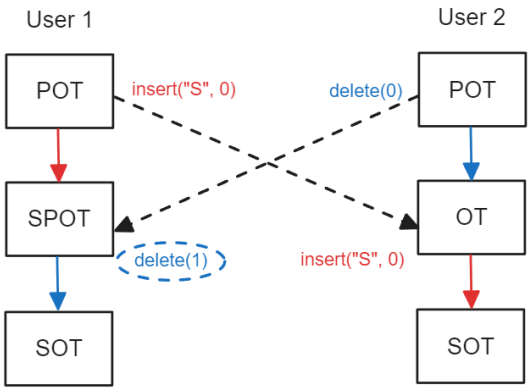

Figure 2.3: Example of the OT algorithm, image based on the Conclave case study [1]

Even though this is a totally functional solution, it has many scalability issues. All transformations have to be made by a stateful centralized server, that needs to be aware of all previous operations in the document, limiting clients into communicating through the same server instance, which can create a bottleneck as the number of users in the same document grows. Besides, the OT algorithm is extremely complex to implement effectively. For those reasons, we decided that OT was not the most viable approach for handling conflicts.

Conflict-Free Replicated Data Type (CRDT)

CRDTs provide an alternative approach to conflict resolution. They are designed such that their operations are always commutative and idempotent, as mentioned previously. Rather than implementing a complex algorithm to handle conflicts like OT, it uses a more complex data structure.

This algorithm provides a variety of advantages over OT: it is easier to implement, it allows users to communicate through different stateless server instances even when in the same document and changes sent to the server by offline clients can be arbitrarily delayed. This makes CRDT a more viable and scalable approach, since it allows a decentralized and efficient architecture.

The CRDT algorithm works by assigning a globally unique identifier to each character in the document, which is a combination of a unique site identifier (unique identifier for each client) and a unique character identifier. Also, the CRDT algorithm requires a way to globally order characters, instead of absolute indices, which depends on the type of CRDT algorithm.

When a user inserts a character, the algorithm generates a new identifier between the neighboring characters. When a user deletes a character, the algorithm simply marks the character as deleted without actually removing it from the data structure, creating what it's called a *tombstone character*. This way, the algorithm can maintain the order of the characters correctly and handle concurrent operations without conflicts.

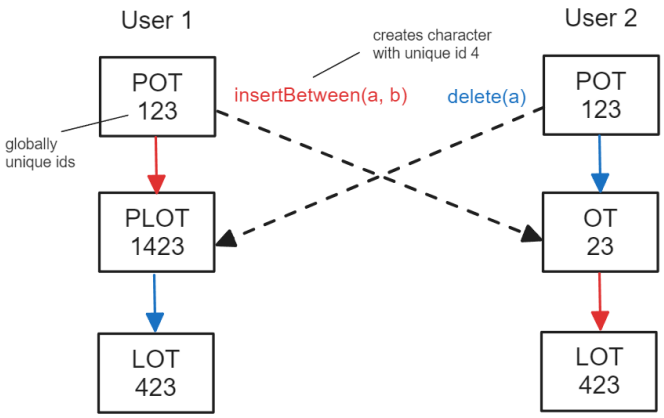

Figure 2.4: Example of the CRDT algorithm, image based on the Conclave case study [1]

Fugue CRDT

There are a lot of CRDT variants, but most of them share the same problem - an anomaly that occurs on both OT and most CRDT algorithms. It is called "interleaving", which can happen when concurrent changes made by different users overlap or are mixed together, with the merged outcome resulting in an unreadable or undesirable state. This is very common in an online/offline scenarios, where a user that is online and other that is offline both insert characters in the same positions. When the offline users reconnects and sends its changes to the server, this anomaly occurs.

The Fugue algorithm is a recent CRDT algorithm that introduces a unique mechanism to handle such anomalies, ensuring the correct merging of concurrent edits, thereby maintaining data integrity even in such cases. *"It is named after a form of classical music in which several melodic lines are interwoven in a pleasing way"* [3]. Fugue is particularly notable for its simplicity and effectiveness in maintaining a "uniquely dense total order" among list elements.

In the fugue algorithm, each character is considered a node in a tree. Each node in the tree, except for the root, has a parent and a side (indicating if it's a left or right child). Multiple nodes can have the same parent and side, making it an n-ary tree. Each node is labelled by a causal dot (the previously mentioned combination between the site id and the character id), and the order of characters is obtained through an in-order traversal: first traverse the node's left children, then visit the node itself and then traverse its right children. If there are ties between children on the same side, they are resolved by sorting alphabetically the causal dots.

There are two ways of implementing the algorithm: a string-based approach or an actual tree structure approach, each with its upsides and downsides. The string implementation uses "string positions" as a proxy for a tree structure, that is implicitly establishes the tree-like algorithm with the identifiers of characters. The traversal of the tree is done simply by alphabetically sorting the characters. In the tree implementation, the tree structure is explicitly maintained, with nodes representing the positions of elements in the list. Each node has a unique label (causal dot) and references to its left and right children.

The string-based implementation has several advantages:

- **Simplicity**: Positions can be compared using standard string comparison, making it easy to implement;
- **Efficiency**: Sorting and insertion operations can be performed efficiently by leveraging string comparison.

However, it also has some downsides:

- **Positional Growth**: As more insertions occur, position identifiers may grow in length very quickly, potentially impacting performance;
- **Imprecision**: The reliance on lexicographic ordering might lead to inefficiencies in densely populated segments of the list.

The tree-based implementation offers a more explicit and flexible structure:

- **Direct Manipulation**: Nodes can be directly manipulated, allowing for more fine-grained control over the structure;
- **Scalability**: The explicit tree structure can handle a high volume of operations without significant performance degradation;
- **Clear Semantics**: The tree structure provides a clear and intuitive representation of the order of elements.

The downsides include:

- **Complexity**: Managing an explicit tree structure requires more complex algorithms and data management;
- **Overhead**: The additional metadata and references required for each node can introduce overhead;
- **Synchronization**: Ensuring that all replicas maintain a consistent view of the tree can be challenging in a distributed environment.

In summary, the choice between string-based and tree-based implementations depends on the specific requirements and constraints of the application. The string-based approach offers simplicity and efficiency for general use cases, while the tree-based approach provides greater flexibility and scalability for more demanding scenarios. Our choice is later mentioned and explained in the chapter 5.

More information about the actual implementations of this algorithm can be found in Matthew Weidner's original web article on Fugue.[4].

Chapter 3

Problem Description

3.1 Problem

Traditional methods of project collaboration often rely on exchanging local copies of documents, which can be cumbersome and inefficient. While tools like Git allow for asynchronous collaborative work and version tracking, they lack the ability to support real-time, synchronous editing of documents. Google Docs addressed this by enabling multiple users to edit the same document simultaneously; however, it lacks robust version control and advanced conflict resolution mechanisms found in systems like GitHub. Therefore, we agreed that there is a need for a comprehensive solution that combines the real-time collaboration and rich text editing of Google Docs with the version control and public sharing capabilities of GitHub.

3.2 Solution

To address this, we propose NoteSpace, a multi-platform web application that facilitates real-time document collaboration and sharing.

3.2.1 Functional Requirements

- **Rich Text Editor**: Allows users to create and edit documents with markdown support;
- **Real-Time Collaboration**: Enables multiple users to edit documents simultaneously, ensuring consistency across all clients;
- **Workspace Management**: Provides organization of documents and folders within workspaces;
- **Workspace Sharing**: Enables sharing of workspaces via links or through workspace discovery in the platform;
- **Cross-Platform Compatibility**: Ensures that documents can be accessed and edited from any device;

- **Offline Mode**: Allows document access and editing even when offline, with synchronization upon reconnection;
- **Version Control**: Tracks changes to documents and allows reverting to previous versions.

3.2.2 Non-functional Requirements

- **Scalability**: The platform should support a large number of users and documents without compromising performance;
- **Performance**: Real-time editing should be responsive with minimal latency;
- **Security**: Ensure secure authentication and authorization to prevent unauthorized access to documents;
- **Reliability**: The system should be robust and handle concurrent edits without data loss;
- **Maintainability**: The codebase should be modular and well-documented to facilitate updates and maintenance.

Chapter 4

Architecture

The system is comprised of two main modules: the backend application and the frontend application, as can be seen in figure 4.1. The backend acts the intermediary between clients and databases, as well as the module responsible for managing real-time updates on every client. The frontend is composed of a web application that provides simple interfaces and user interaction with the system. Communication between both modules is achieved through HTTP[5] for requests that are non-polling, and WebSockets[6] for streamed communication.

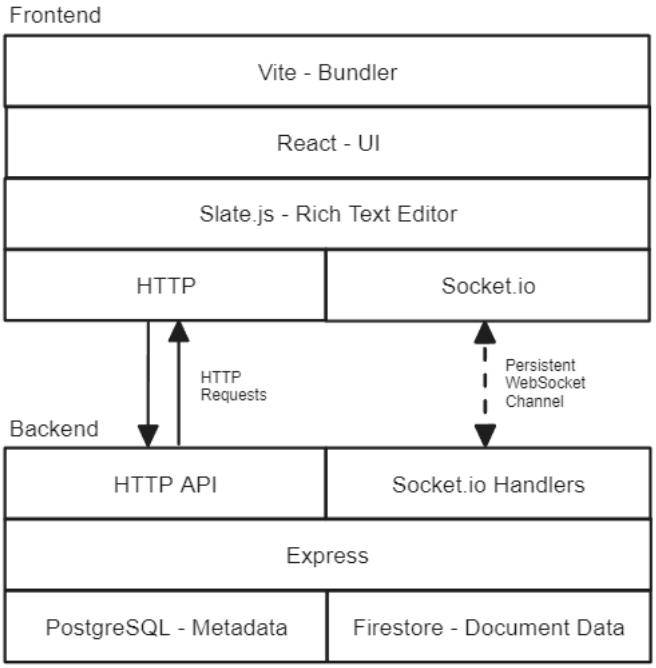

Figure 4.1: System Architecture

4.1 Data Model

System data is divided into two categories: content data and metadata. Content data refers to large chunks of data that represent the content of a given resource, e.g the data needed to describe the content of a resource. To note that content data is not a raw representation of the content of a resource, as it could hinder some processes, such as state consistency across users, which will be further explained in chapter 5. On the other hand, metadata refers to the descriptive information about a resource, namely its identification and relation between other resources. Figure 4.1 also shows the different database modules.

4.1.1 Data Splitting

As content data and metadata serve different roles, each type of data comes with diverging requirements. Content data handling requires fast read/write speeds, specially for highly recurring writes in smaller time-frames, as well as flexibility regarding the data structure for storage. In this regard, No-SQL databases are the ideal choice. In the end, we opted with Firestore[7] for it's simple API as well as the previous experience we had with the database. Similarly, metadata requires strong data structuring, as well as more complex queries. For this reason, we opted for a SQL database, namely PostgreSQL[8]; in addition to the previously mentioned requirements, we took advantage of internal tools such as triggers to automate database-related tasks, removing strain from the server and improving request-processing times.

4.1.2 Metadata

As mentioned previously, metadata provides necessary data to identify and relate datasets, as well as provide any additional information; it's comprised of two main entities: Workspace and Resource. A workspace acts as an analog to a Git repository. Its purpose its to organize multiple related resources to improve ease-of-use and organization for the user. Each dataset is independent of each other, as they have their own resources and related data.

Resources have a similarly function to workspaces, but in a smaller scale. They can act as means to organize multiple related resources, but are not independent from other resources, given that those share the same Workspace, as they keep a parent-child relation, as seen in figure 4.2.

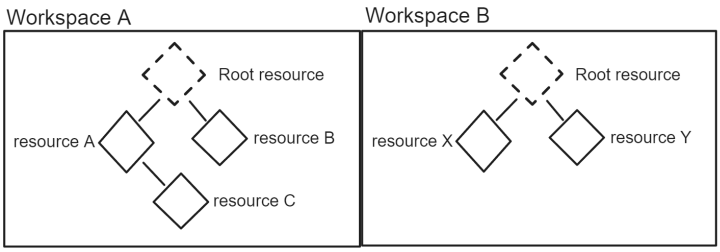

Figure 4.2: Relation between workspaces and resources

4.1.3 Content Data

The main goal of content data is to store content-related data, which can be significantly larger than the metadata that represents it; a simple example of this can be seen with documents. To represent a document while still ensuring state consistency across users, each user-performed operation is stored in an array of operations, instead of the actual document content. This allows the server to be stateless since all it does is provide and store the operations. The document operations are then stored in Firestore, more specifically in a collection that represents the workspace and a document that represents a document. This allows for fast read/write speeds, specially as a large amount of operations can be done in smaller time-frames. Note that folders to not have any data other than the metadata, so the only data stored in the document database are the document operations.

4.1.4 Data Organization

In order to improve read/writes speeds, all data is stored in their most basic form, keeping the minimum data needed to represent them and their relations with other datasets, while the implementation of those relations is only done in client-side. This vastly improves performance, as, for example, representing the tree-structure of resources in a workspace directly in the database would slow all CRUD[9] operations; instead, the tree is only built in client-side when needed. More details can be seen in chapter 5.

Workspace Tree

As previously mentioned, resources can have a parent-children relation; for example, a given folder or resource can have other resources as their children. This allows for greater user freedom, as multiple resources can be bundled together to improve workspace and work organization.

Fugue Tree

In a similar manner, a document can be represented as a tree, more specifically, a fugue tree, as mentioned in chapter 2; what differs from the previous approach is that nodes represent characters, instead of resources. The tree itself establishes the order between the characters by the way of how they should be represented in the document. This means that, based on the traversal algorithm which was explained in chapter 2 and will be further detailed in chapter 5, the order from which characters are inserted differ from their position on the tree. This approach, compared with raw document content representation, ensures consistency between users.

4.2 Frontend

The frontend application acts as the connection between the client and the system. It provides a simple interface and allows users to manage workspaces, folders, resources, and collaborate in real-time with other users. Its composed by a single page application (SPA) with multiple layouts. Each layout is comprised of a multitude of components and the application's service layer is responsible to communicate with the backend through either HTTP[5] or WebSockets[6].

4.3 Backend

The backend is responsible for handling all incoming requests from the clients, processing them to ensure request validity. It also acts as both an intermediary between client and the data layer, and a synchronization-agent, ensuring all connected clients always reach state consistency.

4.4 Technology Stack

In this section, we describe the technologies used throughout the development of this project and why we used them.

4.4.1 Core

The core technology stack of NoteSpace includes the common frameworks, libraries and tools that were used both in the frontend and backend components of the application.

TypeScript

Chosen as the main language of development for both the client and server applications, TypeScript[10] simplified the process of development. With its powerful type-safety, code readability and maintainability is improved. Furthermore it helps to quickly catch errors,

when compared to vanilla JavaScript. It also allowed for building common modules that could be used for both the client and the server, further improving simplicity and helped avoid code repetition.

Node.js

Used as the runtime environment for both the backend and frontend applications, it also comes with great tooling and public libraries through Node Package Manager (NPM)[11], which allowed us to maximize our productivity.

Socket.io

Used to enable real-time bidirectional communication between client and server application, Socket.io[12] provides the necessary tools to build a collaborative application like NoteSpace, with events, rooms and a variety of other useful functionalities, like HTTP long-polling fallback, automatic reconnection and message broadcasting.

4.4.2 Frontend

React

To create a SPA with simple user interface, we choose React[13]. It allows us to create reusable components, making the development process more efficient and the codebase easier to maintain. By leveraging React's virtual DOM and state management, we were able to quickly build a responsive and dynamic user interface that updates seamlessly with user interactions.

Vite

Served as the bundler and development server, Vite[14] offers fast and optimized bundling of the application and as well as hot module replacement. Vite's modern architecture and use of native ES modules in development provides a significant improvement in build performance, reducing the time required for initial builds and incremental updates. It also support many useful plugins, like PWA-support.

Slate.js

For the editor styles, both block(headings, lists, etc.) and inline styles(bold, italic, etc.) we needed a framework that could be flexible while still compatible with React. The most popular frameworks that would allow us to achieve this were Slate.js, Draft.js and BlockNote. Overall, we agreed that Slate.js[15] was a better choice, for its flexibility, plugin system and mobile support. Slate's rich API allowed us to implement complex text editing features and customize the editor to meet our specific needs.

Material UI

Occasionally, we used Material UI[16], that provides a library of pre-made React-compatible reusable components, for a better and easier way to implement user interfaces.

SCSS

Sassy Cascading Style Sheets (SCSS)[17] was used for styling our application. As a CSS preprocessor[1], it provides a more powerful and flexible syntax than traditional CSS, expanding upon the basic language, with features like variables, nested rules, mixins for style-templating, and functions for dynamic styles. With it, we improved the readability and maintainability of the style-sheets.

Vitest

Vitest[18] was chosen as the testing framework for the client application, as it seamlessly integrates with Vite and provides a fast and efficient testing environment.

4.4.3 Backend

Express

Express[19] was used as our framework to build the backend API, providing a robust middleware system, routing, authentication and error handling which allows for a faster development for the RESTful API

Firestore

For storing the document-related content, we decided to use Firestore, a No-SQL database that allows for efficient storing and fetching of smaller-sized documents. It was chosen against other No-SQL databases due to its ease-of-use and simplicity. A No-SQL database was also chosen against a relational database due to its efficiency on larger documents, compared with SQL databases.

PostgreSQL

For storing all metadata, such as user, workspace and document-related info, we used a PostgreSQL database which provides a powerful, reliable and structured database solution, that handles better structured data and complex queries efficiently. It also supports a robust set of tools like triggers that allow for the automation of tasks on the database layer, removing responsibility from the server.

[1] In the context of computer science, preprocessors are used to process data input into some other valid format to be compiled later. SCSS preprocesses files into CSS, so that they can be used in browsers.

Docker

Docker was used to deploy our PostgreSQL database, ensuring a better production environment, better portability and scalability and simplifying deployment and maintenance processes.

Jest

For testing our backend code, we used Jest, which is a powerful testing framework with features such as test runners, assertion libraries and mocking capabilities.

4.4.4 DevOps

Git/GitHub

Git was used for the version control of our project, enabling collaborative development and efficient management of our codebase. GitHub provided a platform for hosting our repositories, organize our tasks in issues and manage our project in a project board.

GitHub Actions

GitHub Actions provided us continuous integration and continuous deployment (CI/CD), by automating our tests in each commit pushed to the main branch, reducing the risk of errors and improving our overall development efficiency.

ESLint/Prettier

ESLint and Prettier were used to enforce quality and consistency across our codebase. ESLint provided static analysis to identify and fix potential issues in our code, while Prettier ensured a consistent style by automatically formatting the codebase.

Chapter 5

Implementation

5.1 Development Methodology

According to standards, as well as a team decision as the most efficient approach for the project, feature implementation was done throughout following the Agile Methodology[20]. This allowed for a continuous implementation process, with each feature getting its own development cycle. We also opted for feature-based Scrums, where one or more sprints would be used to develop a single feature. This required better earlier planning in order to efficiently use the most out of the sprint, instead of week-planning, which could slow down productivity. We also used auxiliary tools, like GitHub Projects for task managing, Whimsical[21] for drafting and Excalidraw[22] for visually planning.

5.2 Markdown Supported Editor

To support document editing, the user firstly needs a way to produce and modify content. This is done through a Markdown-supported text editor. This allowed most supported text operations, as well as simple markdown styles, such as headings, lists, bold, italic, etc.

For this we relied on Slate.js, as it was one of the few React-compatible rich text editor libraries that also allowed for further expansion, which was vital as we needed a way to integrate user operations on the editor with our conflict resolution algorithm, which will be explained in the next section.

Slate.js internal structure is fairly straightforward, as it organizes nodes into Descendants, which are higher-order nodes that each represent a line in a document and information about said line, such as a its style, and Children, which are sub-parts of a Descendant and could represent different parts of a line, such as texts with different inline styles, like bold, italic or underline. For every operation, Slate.js provides data that we then convert into valid cursors. A cursor is made of a 'line' and a 'column' field; when an operation gets executed - slate returns a 'path' and 'offset' properties; the path is an array made of 1 or more entries, being the first one the index of the Descendant, and the second the correspondent Children; the offset is the internal offset relative to the children indicated by the path array. These

properties are then parsed into cursors through a simple algorithm:

Algorithm 1 Slate to Cursor Algorithm

Input: $path : [a, b]$, $offset : integer$

```
cursor ← {a, 0}
i ← 0

nodes ← list of descendant at path[0]
while i ≤ path[1] do
    cursor.column ← i + node[i].text.lenght
    if i + 1 ≥ nodes.length then break;
    else
        i ← i + 1
    end if
end while
```

Output: $cursor =$ `{line, column}`

The way we integrate slate with our conflict resolution algorithm will be fully detailed in section 5.3.1.

5.3 Real-time Collaboration

In order to maintain documents states' consistency across all users, we took advantage of a conflict resolution algorithm, most specifically Weidner and Kleppman(2023)'s fugue approach. Our first version was based on their "string implementation", which initially seemed promising, since the order of the characters was established lexically, removing the need of explicit tree entities entirely. However, we soon realized that it had many drawbacks, mostly regarding the complexity of the algorithm for creating the string positions, code maintainability and future compatibility with other features, such as supporting markdown styles. Moreover, scalability was also a problem, since we also observed that the size of string positions would get exponentially larger over time.

Tree

For these reasons, we opted with a version based on Weidner and Kleppman's second approach [23], which more closely resembles the original proposed idea [3]. In this implementation, a class called FugueTree was implemented to represent the tree itself. The class consists of two main properties: root and nodes; the root represents the node from which every other node is a child, and acts as the starting spot for tree traversal. It also stores other details which will be further explained in section 5.3.3. The 'nodes' property is a map that maps a user's id to all nodes inserted by them. All nodes, including the root node, have the following properties:

- **id** - a causal dot between a replica id (explained in sub-section 5.3.1) and the node's counter;
- **value** - represents the value held by the node;
- **isDeleted** - whether the node is a tombstone character (see section 2.1.2);
- **parent** - the parent's identifier;
- **side** - the node's side relative to the parent;
- **leftChildren** and **rightChildren** - array of children identifiers;
- **depth** - used for optimized tree traversal (see section 5.3.3);
- **styles** - an array used to store the node's current Markdown styles (see section 5.3.1).

In order to get the document's traversed state(i.e. the order by which characters are displayed to users), we use fugue's in-order traversal algorithm. The algorithm is non-recursive as it could cause an overflow on the stack for larger trees.

5.3.1 Frontend

As previously mentioned, users produce text by interacting with the text editor, but in order to integrate both the interface and the conflict resolution algorithm, we need a wrapper that acts as an intermediary between the tree and the editor. This intermediary wraps around the tree in order to support text-based operations, expanding upon the tree's general structure. We called this wrapper Fugue and the flow between the two modules is as follows:

1. User triggers an operation on the editor;
2. This, in turn, triggers one of Slate.js many events that could either be a wrapper around vanilla DOM events or not;

3. Slate calls, through intermediary operation handlers, a fugue method which will modify the tree in order to represent the editor's current state;

4. The intermediary handler will call the service layer as well, in order to update the server on the new operations.

The use of intermediary handlers allows for a two-way decoupling, one between the editor and the Fugue wrapper, as well as between the wrapper and the service layer. This allows for better testability and isolation of components.

Its also in the wrapper that cursor-node translation occurs. The wrapper traverses the tree until the implicit cursor matches the target destination. This can also be extrapolated into two cursors(a selection), as operations such as deletions can require the 'deletion' of multiple nodes.

Other than wrapping the tree and providing extra functionalities, the wrapper class provides additional information used by the tree, such as a replica id. This id is unique not only amongst different users, but also amongst different devices/sessions on a single user. This allows to differentiate the same user when using multiple devices.

In terms of the tree, we also choose to store the styles inside it, in two different ways. The first type, which are inline styles like bold, italic and underline, are stored inside each node, as they apply individually. The second type, which are block styles like Headers and List-item, apply to whole lines; for this reason, and to avoid the usage of extra data structures, we choose to save all block styles in the root node in its 'styles' field, where the line on which a given block style applies is their direct index on the array.

Finally, the service layer is responsible for transmitting the operations to the server. This could be done with a operation-based granularity, where each operation would mean a different transmission, but such approach would result in a massive usage of network resources for a small set of operations, as well as a transmission speed which could overload the server and cause inconsistencies on the data layer, as databases such as Firestore would struggle to process vast amount of write operations in such a short time-frame. The solution was then to implement an operation buffer. This buffer would store operations and later send them trough the web-socket stream when one of two conditions were met:

1. The size of operations stored reached the max chunk-size; all operations would be sent in the same stream, instead of multiple streams, as a chunk. This allowed for a better usage of network resources;

2. The chunk-size was not reached but the buffer internal timer triggered a timeout; this still allowed for the bundling of operations even in cases where the max-supported chunk-size was not reached, which allows to quickly stream them without waiting for the chunk to fill-up with operation that may take longer to occur, which in turn could cause state inconsistencies between clients.

5.3.2 Backend

We previously mentioned how batches of operations are sent to the server, but not how they are structured. Instead of simply sending the node or the cursor where to execute the operation on, we send the absolute position of the nodes to modify, we send the necessary information needed to execute the operation while throught the use of absolute ids for

- Sending node data and parent id in insertions
- Sending nodes' ids that must be updated or deleted

By doing this, we ensure that operations are always executed on the correct position, instead of relying on cursors that are constantly changing.

The server receives and broadcasts the received operations as they are, ensuring no state is needed and following one of CRDT's principle, which is that every client receives the same sets of operations. This means that every client, when receiving new remote operations, must apply them internally, which is also done through the Fugue wrapper. Finally, the backend stores the operations on the content-data database as they come, as opposed to storing a copy of the Fugue tree, as this extra step would require that the server kept an internal copy of the tree, modify it on every operation, and finally store it on the data layer. For a multitude of documents opened, this would cause a bottleneck, so instead the backend stores the operations in the order they are received, and the processing needed to convert them into the equivalent tree is deferred to the client-side.

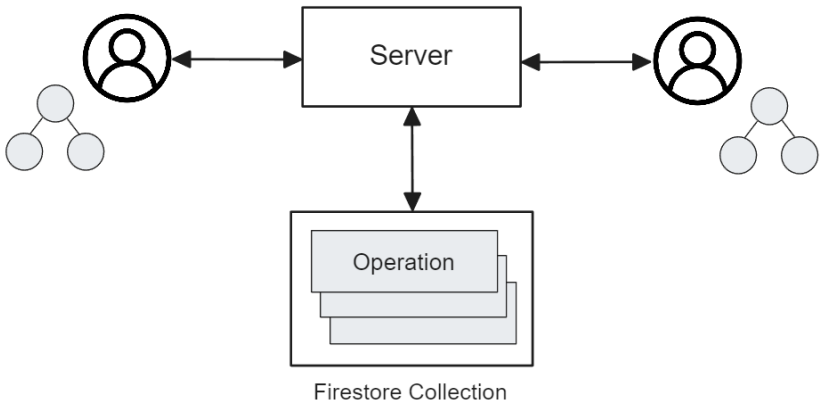

Figure 5.1: Simple example of client trees and data storage

5.3.3 Optimizations

In order to improve traversal, two main optimizations were added to the basic structure. The first one was adding the 'depth' property on the basic node structure. This allows to early checking if traversal can be done on a node's child, instead of only doing that during

the loop's iteration. The second optimization was made by indexing lines' root nodes. As mentioned in section 5.3.1, operations are cursor-based; this means that, in order to translate a cursor into a valid node, we need to traverse the tree. This can cause a bottleneck if a full tree traversal is needed for every single operation so, by indexing nodes which represent line breaks(nodes with character value of '`\n`', which we call line root nodes), the traversal cost for an operation will significantly decrease. Let's suppose the tree is comprised of 20 lines, each with 30 characters. Now let's suppose we want to insert a new character in line 20 at index 10, resulting in the following cursor: (20,10). In the old version, we would need to traverse the whole tree until we reached that position.

This would mean the following:

```
19 lines * 30 characters + 20th line * 10 characters = 580 iterations
```

Now, by indexing lines' root nodes, the cost would be the following:

```
20th line * 10 characters = 10 iterations
```

As we can see simply by indexing lines' root nodes, we get a better performing traversal algorithm. All this allowed for an algorithm that allowed not only state consistencies, faster processing speeds both in the client and server-sides, as well as capabilities for future features, such as offline support, which trough the use of Weidner and Kleppman(2023)'s algorithm, requires minimal modification due to their resistance against interleaving.

5.3.4 Isolation of Real-time Updates

We previously discussed how the service layer sent all user-performed operations to the backend, which in turn updated all users who were modifying the same document. As mentioned in chapter 5.4, we also support workspaces, which can have documents and folders. For this we must also allow for real-time updates on workspace operations like resource creation, editing and deletion; we first need to explain how we can still use the same WebSocket based communication channel while still being able to update clients with certain kind of message broadcasting filtering, as users on a certain document don't want to get updated on other documents' operations, and all users on a workspace want to get updated on every resource creation, edit and deletion. This can be done by taking advantage of Socket.io's rooms[24], which we can imagine as, for example, chat groups on messaging apps - users can only receive messages on rooms they joined. We implemented two types of rooms: workspace rooms and document rooms. When a user navigates to the workspace page, they must join the respective workspace room. Similarly, upon entering a document, they must join the corresponding document room. When the user leaves these pages, the client must exit the respective rooms.

5.4 Workspace Management

Similar to other platforms' approaches, resources on a workspace have hierarchy between them, as users can nest documents or folders inside another folder, for better organization. This means we must not only keep information about the relations between different resources, but a way to visually represent them to the user.

5.4.1 Backend

All data regarding resources are stored as metadata on PostgreSQL, except for data concerning document content, which is stored on Firestore as explained on section 5.3.2.

Firstly we represent the workspace itself, an entity with two main attributes: id and name which are used to identify them. Next we must represent resources in a way that not only we can associate them with a workspace but be able to represent relations between different resources, as previously mentioned. For this reason, resources are represented by a unique id, a workspace id, a name, a type (document or folder), a parent and an array of children ids.

Through the 'parent' and 'children' fields, we can relate different resources in order to represent hierarchy.

Finally we must implement some rules on resource creation, editing and deletion, as they are crucial in order to keep data consistency throughout; the rules are as follow:

1. When a parent resource is deleted, all children resources are also deleted;
2. When a resource is added as a child of another resource or moved to another resource, both the new and the old parent's children field must be updated with the id of the new resource;
3. When a workspace is deleted, all resources within that workspace must also be deleted.

We can assure point 1 and 3 through the use of foreign keys, but point 2 cannot be assured through simple table creation. We consider resource insertion in the context of two different states of the parent. In one state, the parent is null, indicating that the resource is either the root node[1]of a workspace or a child of the root node. In the other state, the parent is non-null, signifying that it represents a valid resource.

For both situations, after the resource insertion and parent's id validation, we update the corresponding parent by appending the node's id. In case the node already existed but its parent field was modified, such as the result of moving a resource into another parent, we also remove the node's id from the old parent's children field.

5.4.2 Frontend

In the frontend, the state management of workspaces is handled by the workspace context. This context provides a single source of data accessible by any component within the workspace provider. Consequently, when a real-time update occurs, all components using this data are updated, ensuring consistency.

The management of the workspace tree is done via the useWorkspaceTree hook that uses the useState hook to store resource nodes. This ensures that updates are reflected in the UI and provides the necessary operations to add, remove, update, and move nodes in the tree.

[1]The root resource node is created automatically when a workspace is created; its id matches the workspace id for fast querying in other operations. This serves as a way to be able to represent root-level resources on the hierarchy tree.

The nodes in this tree are then used in the Sidebar component to organize documents and folders in a file system-like structure. To convert these nodes into a tree object, we used a Depth First Search (DFS) approach to build the tree object by traversing the children of each node.

Algorithm 2 Get Tree Algorithm

Input: $id : string$, $nodes : Map < string, Resource >$
  $node \leftarrow nodes[id]$
  $children \leftarrow []$
  for $childId \in node.children$ **do**
    $children \leftarrow children \cup \text{getTree}(childId, nodes)$
  end for
Output: tree = {node, children}

The rendering of resources in the tree is then done recursively using the TreeResourceView component, that represents a single resource (either a document or a folder) and its children, which are also rendered as TreeResourceView components recursively. Note that not only folders can have children, but also other documents. This design choice allows users greater flexibility in organizing their documents.

To move a resource, the user can simply drag it onto another resource, thereby making it its parent. This was achieved by making the resources in the tree draggable and using the drag and drop events of the DOM. To get the resource id of the dragged element, the DOM id had to match the resource id, making this operation possible.

To create a new resource in the tree, the user clicks the plus button at the desired location, triggering a popup where they can choose to create either a document or a folder. They can then rename the document, and if it's a document, click it to navigate to the document page.

Both the Workspaces and the Workspace components use a common component, called DataTable. This reusable component displays content in a table format with built-in functionalities, such as sorting elements by columns (ascending or descending), selecting or deselecting rows, and creating or deleting resources, with the handlers passed as arguments defining these behaviors.

Chapter 6

Testing

This chapter outlines the testing process undertaken to ensure the reliability and functionality of the project. It details the methodologies employed, including unit tests, integration tests and mocking.

6.1 Automated Testing

Automated testing is essential in any software development project, as it enables developers to quickly identify and resolve issues in their code. In our project, automatic tests were executed either through the CLI before pushing updates or by the CI/CD pipeline to automatically test parts of the codebase. We primarily used unit tests, supplemented by some integration tests. Additionally, we employed mocking and other techniques, such as abstracting the interface by which two components communicate. An example of this can be seen on the operation handlers described in section 5.3.1. Another example is the usage of an abstract Communication class, which abstracts the interface used to communicate with the service layer, for both HTTP and WebSocket communication, which in turn communicates with the backend. This allows for mocking the service layer in components that use it, and test them without worrying about side effects from the service layer. This allows for faster testing cycles as well as the isolation of modules.

In the backend, Jest was chosen as our testing framework, used for both unit and integration tests, while in the frontend, we used Vitest for unit testing, namely for testing the domain logic and the React Testing Library for user interface tests.

6.1.1 Memory Module

The memory module is a specialized component designed solely for testing purposes. It simulates the behavior of an actual memory system, instead of communicating with the production databases, allowing efficient testing of the rest of the system without unnecessary overhead. This also allows us to test and validate the project without the database setup, which is useful in the CI/CD environments, like GitHub actions.

Chapter 7

Project Overview

This chapter provides an overview of the work completed and outlines the future work planned to further expand the project's functionality.

7.1 Completed Work

In the last 4 months of development, the project has evolved significantly since the initial proposal. The most significant milestones completed were:

- **Conflict Resolution** - fugue algorithm;
- **Text Operations** - insert/delete text, copy, cut, paste, etc;
- **Style Operations** - markdown styles with Slate.js;
- **History** - undo and redo operations;
- **Live Cursors/Selections** - live cursor tracking and selection;
- **Workspace Management** - Workspace CRUD operations, document CRUD operations, workspace tree.

7.2 Future Work

The most significant milestones yet to complete are:

- **PWA** - Offline access, data-syncing with offline users;
- **Sharing Platform** - Workspace discovery, searching, etc.;
- **Authentication** - OAuth 2.0 authentication, user profiles, etc.;
- **Version Control** - Changes commit & rollback, version history;
- **Final Touches** - Mobile UI/UX, optimizations, bug fixes.

Bibliography

[1] Sun-Li Beatteay Nitin Savant, Elise Olivares. Conclave - a private and secure real-time collaborative text editor. https://conclave-team.github.io/conclave-site, 2023.

[2] Phoenix NAP. How does git work? https://phoenixnap.com/kb/how-git-works, 2021.

[3] Matthew Weidner and Martin Kleppmann. The art of the fugue: Minimizing interleaving in collaborative text editing, 2023.

[4] Matthew Weidner and Martin Kleppman. Fugue: A basic list crdt. https://mattweidner.com/2022/10/21/basic-list-crdt.html, 2022.

[5] mdn web docs. An overview of http. https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview.

[6] mdn web docs. Web sockets. https://developer.mozilla.org/pt-BR/docs/Web/API/WebSockets_API.

[7] Google. Firestore: Nosql document database. https://cloud.google.com/firestore?hl=en.

[8] The PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org.

[9] Codecademy Team. What is crud? urlhttps://www.codecademy.com/article/what-is-crud.

[10] Microsoft. https://www.typescriptlang.org.

[11] NPM Inc. https://www.npmjs.com.

[12] Socket.io. 1urlhttps://socket.io.

[13] Meta Open Source. https://react.dev.

[14] Vite—next generation frontend tooling. https://react.dev.

[15] Ian Storm Taylor. https://github.com/ianstormtaylor/slate.

[16] Material UI SAS. https://mui.com.

[17] Sass. https://sass-lang.com.

[18] Matías Capeletto Anthony Fu. https://vitest.dev.

[19] Open JS Foundation. https://expressjs.com.

[20] Sarah Laoyan. What is agile methodology? (a beginner's guide). https://asana.com/pt/resources/agile-methodology, 2024.

[21] Whimsical. https://whimsical.com/.

[22] Excalidraw. https://excalidraw.com.

[23] Matthew Weidner. Tree fugue. https://github.com/mweidner037/uniquely-dense-total-order/blob/master/src/implementations/tree_fugue.ts, 2023.

[24] Socket.io. Rooms — socket.io. https://socket.io/docs/v4/rooms/.

[25] Mehul Ghala. The enigma of collaborative editing, 2019.

[26] Matthew Weidner. Fugue: A basic list crdt. https://mattweidner.com/2022/10/21/basic-list-crdt.html, 2022.