Distributed Systems Project Report: P2PDocs Building a Decentralized Google Docs Clone

Minisini, Alessandro
158658@spes.uniud.it

De Biasi, Alessandro
157698@spes.uniud.it

Piccin, Nazareno
170521@spes.uniud.it

GitHub Repo

Abstract

This document aims to describe the design, implementation, and testing methodology of P2PDocs, a peer-to-peer collaborative text editor developed for the Distributed Systems course at the University of Udine.

This report serves both as a technical documentation of the system and a guideline for its development, highlighting key challenges in decentralized coordination, conflict resolution using CRDTs (Conflict-Free Replicated Data Types), and fault tolerance.

While concise, this report covers the core architectural decisions, module interactions, and lessons learned during the project.

The project exemplifies practical applications of distributed systems principles, including eventual consistency, dynamic peer discovery, and fault tolerance in a decentralized environment.

Chapter 1

Introduction

Glossary

Term	Definition
Peer	Istance of the P2PDocs on a machine
System	Abstract concept of the project
Network	Graph of connected peers
Node	Synonym of Peer, specific to Elixir context

Table 1.1: Glossary

Application Description

P2PDocs is a decentralized collaborative text editor that implements a serverless architecture, eliminating single points of failure inherent in client-server models. The system enables real-time concurrent editing through direct peer-to-peer communication in a dynamic network topology.

Users should be able to edit the text even while offline, and the system should automatically resolve emerging conflicts once the connection is re-established.

Distributed Algorithms

We want to design a system that is Eventually Consistent + Available + Partition-tolerant. To achieve that, we will make use of various broadcast algorithms (Echo-Wave and Causal, depending on the context) and CRDTs (Conflict-Free Replicated Data Types): a data structure that allows peers to update concurrently and independently, resolving automatically any inconsistencies that might occur and ensuring that eventually all peers will converge to the same state.

Logical Model Assumptions

The following assumptions have been made:

1. Arbitrary Dynamic Topology
2. Non-Adversarial Non-Anonymous Crash-Recovery Nodes
3. Fair-loss channels
4. Asynchronous Network
5. Possible, but transient, partitions

Chapter 2

Analysis

In this chapter, we describe in detail the functional and non-functional requirements of a solution for the problem.

2.1 Functional Requirements

The following functional requirements define the expected behavior of the P2PDocs system.

Document Editing

- **FR1.1:** The system shall allow users to insert and delete characters in a shared text document.
- **FR1.2:** The system shall propagate local edits to connected peers in near real-time.
- **FR1.3:** The system shall apply remote operations in a manner that preserves causal consistency.

Peer-to-Peer Communication

- **FR2.1:** The system shall establish peer-to-peer connections between clients without relying on a centralized server.
- **FR2.2:** The system shall allow peers to join and leave the network at runtime.

Conflict Resolution

- **FR3.1:** The system shall resolve concurrent operations automatically.

- **FR3.2:** The system shall ensure eventual consistency, meaning all peers converge to the same document state given sufficient connectivity and time.

Crash Recovery and Synchronization

- **FR4.1:** The system shall persist the local document state and unsent operations locally to enable recovery after crashes.
- **FR4.2:** Upon reconnection, the system shall synchronize the local state with that of a connected peer.

Dynamic Membership

- **FR5.1:** The system shall support the integration of new peers at any time during the editing session.
- **FR5.2:** New peers shall obtain the current state of the document upon joining the network.
- **FR5.3:** The system shall allow peers to disconnect without impacting the editing session.

User Interface

- **FR6.1:** The system shall provide a basic graphical user interface (GUI) for text input and document visualization.
- **FR6.2:** The GUI shall update in near real-time to reflect local and remote edits.

2.2 Non-Functional Requirements

The following non-functional requirements define the quality attributes of the P2PDocs system.

Availability

- **NFR1.1:** The system shall be tolerant to individual client failures without affecting the continued operation of the network.
- **NFR1.2:** The system shall allow disconnected peers to reconnect and resynchronize with minimal user intervention.

Scalability

- **NFR2.1:** The system shall support editing sessions with a variable number of participants.
- **NFR2.2:** The system's communication and merge overhead shall scale *reasonably* with the number of concurrent operations.

Consistency

- **NFR3.1:** The system shall guarantee *eventual consistency* across all non-faulty, connected peers.
- **NFR3.2:** It must be ensured that a deterministic merge behavior under concurrent modifications.

Fault Tolerance

- **NFR4.1:** The system shall tolerate peer disconnections and automatically attempt to re-establish connections upon availability.
- **NFR4.2:** In the event of a crash, a peer shall restore its document state using persistent local storage.

Usability

- **NFR5.1:** Users shall not be required to resolve merge conflicts manually.

Chapter 3

Project

This chapter is devoted to the description of the general architectures and specific algorithms.

3.1 Logical architecture

The Project is composed of two main parts: the frontend and backend.

Frontend

The front-end consists of two simple modules: a graphic interface for user interaction and a module that enables communication between the former and the backend.

UI Rendering Module

The UI Rendering Module constitutes the presentation layer of P2PDocs. It is tasked with visualizing the shared document state and reflecting real-time updates received from the backend infrastructure. This module is built to interface with the API Client Module, which manages communication with the Elixir-based backend.

API Client Module

The API Client Module serves as the communication gateway between the frontend application and the Elixir-based backend. Its primary function is to abstract over network protocols, enabling the application to perform both real-time and request-response interactions with backend services in a decoupled and resilient manner.

Backend

The backend is more complex, indeed, it is where most of the logic is implemented, and it is formed by several modules:

API Module

The Backend API Module functions as the primary interface for handling communication with the frontend. It exposes a set of bidirectional communication channels designed to receive client-issued commands and dispatch asynchronous event-driven updates. This module is responsible for interpreting incoming messages, invoking the appropriate business logic, and broadcasting state changes to relevant connected clients.

CRDT Module

Manages document operations using an Operation-based Conflict-free Replicated Data Type (CRDT). Applies local edits and integrates remote operations in a causally consistent order, ensuring eventual consistency across replicas.

Auto-Saver Module

This module auto-saves the document state after a pre-set number of operations. It retrieves the state from the crdt, and it saves it with Elixir's built-in functions to manage files.

Supervisor Module

The Supervisor Module is responsible for observing critical backend components such as the Network Modules and the CRDT Module. It acts as a supervisor that detects failures and restarts the module accordingly. It is also responsible for storing the states of all the modules so that they can be restarted with the previous configuration.

Network Module

The Network Module, composed of the two submodules Causal Broadcast and Echo Wave, abstracts peer-to-peer networking functionality, ensuring causality between messages. It interfaces with the Reliable Transport Module for message propagation.

Neighbor Handler Module

The Neighbor Handler Module manages peer discovery and network membership. It is responsible for handling the joining/leaving of peers, responding to whole-document state requests, and initiating state synchronization for newly connected nodes.

Reliable Transport Module

The Reliable Transport Module is responsible for implementing a reliable communication for propagating state updates and events across participating nodes, without duplication.

3.2 Protocols and algorithms

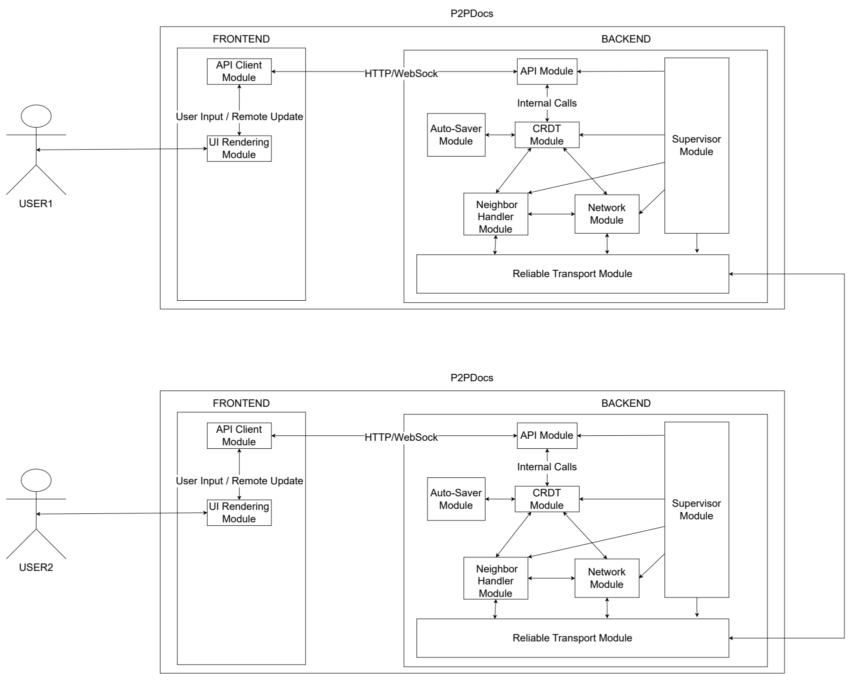

Figure 3.1: Modules diagram

Data model

The text editor must provide two main operations to users: *insert* and *delete*.

Insertion

We aim to ensure the property of *intention preservation*, a concept also present in Operational Transformation algorithms, such as those used by Google Docs.

Definition 1 (Intention Preservation)**.** *If a peer inserts the character x between the characters a and b ($a \prec x \prec b$), this relation must be preserved in every replica across all peers.*

In simpler terms, we need to convert a relative position into an absolute one in such a way that the above property holds.

A naive approach is to use *rational* numbers instead of integers for positions. Since $(\mathbb{Q}, \leq)$ is a dense linear order, it is always possible to find a rational number x between any two rational numbers a and b (e.g., using the mean $\frac{a+b}{2}$). Moreover, each fractional position can be interpreted as a path in an ordered tree: each digit represents the index of a child, and the number of digits indicates the depth in the tree.

However, we want to avoid using too many bits for a single position. Formally, the number of bits used should be sublinear with respect to the number of positions generated. This means we need to control both the depth and fan-out of the implicit tree we construct. A more sophisticated approach that addresses this issue is the LSEQ strategy [NMMD13], which provides a simple and efficient representation.

While these solutions ensure intention preservation, they do not guarantee that positions generated by different peers are unique. To resolve this, each position digit can be tagged with the identifier of the peer that created it. In case of conflicts, ties are broken using the lexicographic order of the peer IDs.

Deletion

The same character may appear multiple times in a text, so we cannot identify each character solely by its value. Therefore, we need a globally unique identifier for every character inserted by a peer. This allows us to issue a delete operation by simply referring to the character's identifier.

It is important to note that we cannot use the absolute position of a character as its identifier, since the same position might be reused after the deletion of that character.

To generate globally unique identifiers, we can concatenate the peer ID with a local counter that is incremented each time the peer inserts a new character: in this way, we can avoid using consensus algorithms and slow down the entire system.

To summarize, each character will be represented by a triple $\langle ID, pos, value \rangle$ where

- *ID* is a string, composed of the peer ID and a local unique ID;
- *pos* is a vector of integers generated by the LSEQ strategy, representing the absolute position of a character that preserves the relation described in Definition 1;
- *value* represents the actual character.

CRDT

We will use an Operation-based Conflict-Free Replicated Data Type, implementing the operations described in the data model.

Broadcasting information

We will mainly make use of two broadcasting protocols: Echo-Wave and Causal. The former will be used at the "transport" layer to make sure that each peer gets notified of the operation made by the initiator. The latter will be used at the "application" layer to make sure that each delete and insert operation on the same indexed character is processed in the correct order.

Joining and Leaving the network

When a peer joins the network, it notifies its neighbors to receive all the necessary information to sync its state. When a peer leaves the network, it notifies its neighbors to allow the network topology to remain a single connected component.

3.3 Physical architecture and deployment

The back-end is written in Elixir and is natively executed over the Erlang virtual machine (BEAM). The front-end, instead, is written in Node.js and Express. The overall project is run inside a Docker container for portability and usability.

3.4 Development plan

Tier 1

- No GUI and local storage
- Working under static topology and crash-free nodes assumptions

Tier 2

- No GUI and local storage
- Working under dynamic topology and crash-free nodes assumptions

Tier 3

- No GUI and local storage
- Working under original assumptions

Tier 4

- Full requirements fulfilled

Chapter 4

Implementation

The back-end modules are developed using Elixir, a functional programming language running on the Erlang Virtual Machine. The front-end module is built in JavaScript and communicates with the back-end via WebSockets. The entire software package is encapsulated within a Docker image, facilitating easy portability and deployment across various platforms.

Below are detailed notes on individual module implementations. Interested readers are encouraged to read the project's documentation, which provides comprehensive specifications for modules and functions.

4.1 Supervisor

All Elixir modules operate under a Supervisor, responsible for restarting modules that crash and preserving their state using Erlang Term Storage (ETS). This approach ensures fault tolerance, enabling peers to recover seamlessly and resynchronize with others following a crash.

4.2 CRDT Structure

The CRDT is implemented using an AVL order-statistics tree (OSTree). This structure enables efficient insertions, deletions, and positional queries, all in $O(\log n)$ time. Besides standard operations, the tree efficiently supports retrieving the k-th smallest element and finding the index of specific elements, facilitating quick responses to front-end queries and updates.

The CRDT follows an operation-based approach inspired by the LSEQ allocation strategy, dynamically assigning positions between adjacent characters. Positions are represented as ordered lists of `(digit, peer_ID)` pairs, ensuring a unique, conflict-free order across distributed peers.

Invariant checks guarantee that newly assigned positions adhere to logical ordering constraints, thus preserving users' editing intentions.

Interaction with other layers and the front-end is managed by a GenServer—an Elixir behavior designed to handle synchronous and asynchronous requests. This GenServer also coordinates the Auto-Saver module, which saves document edits to a local file.

4.3 Network Layers

All network modules rely on asynchronous calls managed through GenServers. The primary modules utilizing this behavior are CausalBroadcast, EchoWave, ReliableTransport, and NeighborHandler.

EchoWave

The EchoWave module manages active waves, each corresponding to a CRDT operation initiated by a user or received from another peer. Each wave is assigned a unique identifier – the vector clock of the initiator – for tracking.

NeighborHandler

The NeighborHandler module manages peer connections and disconnections. Peers specify the unique identifier of the peer they wish to join through an asynchronous call. Upon connecting, the joining peer receives the current state of the existing peer, including the CRDT, vector clock, and causal broadcast delivery clock. This mechanism ensures synchronization and consistency among peers.

NeighborHandler also ensures network integrity during peer disconnections. To avoid network partitioning and inconsistencies, departing nodes must inform neighbors and share their neighbor lists. Neighbors then form a clique to maintain connectivity, which could have been disrupted by the node's exit.

ReliableTransport

The ReliableTransport module implements a classic send/acknowledgment mechanism to guarantee message delivery in case of a node crash. A configurable timeout, defaulting to 5 seconds, determines resend behavior. Additionally, the module stores identifiers of received messages, preventing duplicate deliveries.

4.4 Front-End

The front-end supports basic text editing operations, specifically inserting and deleting individual characters. The document is treated as a single line, and newlines are seen as characters. We support basic text operations such as selection, copy/paste/cut, but not replacements. Built primarily in JavaScript with a user interface styled using TailwindCSS, the front-end utilizes Express, a Node.js-based framework, for application logic and server management.

Chapter 5

Validation

Unit tests for each module are implemented with Elixir's built-in `ExUnit` framework and, where needed, the `Mox` library for mocking inter-module calls. To run them, execute:

```
cd p2p_docs_backend/
mix test
```

The output will indicate which tests passed and highlight any failures.

Multi-Node (System-Wide) Tests

End-to-end testing across multiple instances is performed via custom scripts rather than a built-in framework.

`tests/start_n_instances_for_tests.sh <n>` Launches `n` containers of P2PDocs, all attached to the dedicated Docker network `172.16.1.0/24`. Each node's frontend is exposed at `http://localhost:<3000+idx>`, where `idx` is the node identifier between 1 and `n`.

In the `tests/` directory:

- `disconnect.sh <idx>` Disconnects peer `idx` from the network.
- `connect.sh <idx>` Reconnects peer `idx` to the network.

To inspect logs for the peer `idx`:

```
sudo docker logs test<idx>
```

Chapter 6

Conclusion

The P2PDocs project has analysed the viability of a decentralized collaborative text editing system, effectively leveraging distributed systems principles such as eventual consistency, fault tolerance, and dynamic peer discovery. By utilizing CRDTs and advanced broadcast protocols, the system ensures efficient real-time collaboration and automatic conflict resolution, closely replicating key functionalities found in centralized applications like Google Docs.

Through the course of this project, significant lessons were learned regarding the complexity of decentralized coordination, the necessity of robust fault-tolerant mechanisms, and the importance of scalable communication protocols in distributed systems. The implementation has validated the chosen design decisions, notably the operation-based CRDT with the LSEQ strategy, and showcased the efficacy of Elixir and the Erlang virtual machine in building fault-tolerant, highly available applications.

6.1 Future Developments

To further align P2PDocs with the feature set of Google Docs and enhance user experience, several developments are recommended:

1. **Markdown Support:** Enable users to write documents using markdown syntax, allowing intuitive formatting and streamlined document creation.
2. **Real-time User Presence and Cursor Positioning:** Integrate visibility of peer cursors and editing activity, facilitating better collaboration awareness among users.
3. **Version History and Document Snapshots:** Introduce a mechanism for maintaining document versions, allowing users to revert or compare previous document states, thus providing additional reliability and transparency.

4. **File Management and Sharing:** Implement functionalities for document creation, deletion, offering users control similar to traditional document services.

5. **Improved Large Edits Synchronization:** Extend the front-end and crdt logic to manage large-scale edits and ensure faster synchronization, enhancing system smoothness.

6. **Improved Message Complexity:** Since the EchoWave produces a spanning tree, it is worth trying to exploit it to send fewer messages to deliver operations.

7. **Enhanced Vector Clock:** Reduce the size of each vector clock sent, exploiting u-vector and s-vector information, as seen during classes.

8. **Investigate techniques for achieving k-connectivity:** Strengthening the network's connectivity enhances fault tolerance by minimizing the likelihood of partitions.

Bibliography

[NMMD13] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. Lseq: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng '13, page 37–46, New York, NY, USA, 2013. Association for Computing Machinery.