Università di Pisa

Master Of Science in Computer Engineering

Distributed Systems and
Middleware Technologies

Collaborative text editor

Project members:
Francesco De Lucchini
Salvatore La Porta

Academic Year 2025/2026

Contents

1 Introduction

1.1 Project description

Our project idea is to create a (simplified) collaborative text editor, akin to a basic version of applications like *Google docs* or *Overleaf*.

The main functional requirements are that users can create, delete, and, most importantly, **share** notes. Invited users can edit the shared note concurrently: they can view, in real-time, what every other user is typing and the position of their cursor.

The main challenge of this project is managing the consistency of concurrent edits without locking the document, ensuring everyone **eventually** sees the same exact text.

We also focus on horizontal scalability and redundancy: the server must be distributed across multiple nodes, and the state of each note must be replicated among them.

Due to the inherent complexity of collaborative text editing, we adopt the following simplifying assumptions:

1. The editor only supports plain Unicode characters; there is no metadata for text styling (e.g., font, color, size)
2. Users can only input or delete one character at the time (this simplification can be easily relaxed, as an edit of N characters can simply be split into N single-character edits)
3. There is no user authentication mechanism and, consequently, no access control on the notes (again, the focus of this project is not security)

2 Architecture

2.1 Overview

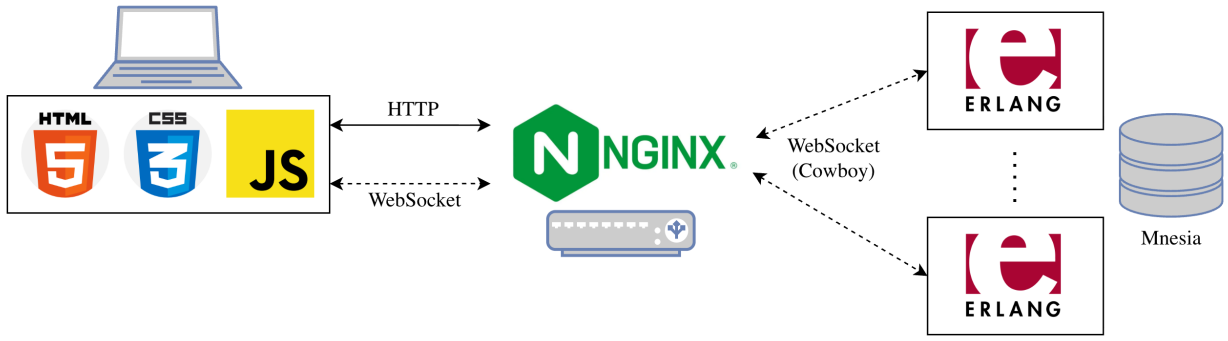

Figure 1: Schema of the software architecture

We now briefly explain the responsibilities of the various components, and later dive deep in each one.

The **web application** implements the user interface and manages the creation, deletion, and sharing of notes. It consists of static HTML, CSS, and JS files served via HTTP by a web server (**Nginx**, in this case).

To send and receive real-time edits on a shared note, the web application communicates with the back-end via **WebSockets**. This communication is not direct; it passes through a load balancer (also Nginx) to efficiently distribute traffic across the multi-node back-end.

The back-end is written in **Erlang**: it stores the state of the notes using **Mnesia** (which is replicated across all nodes) and is also responsible for broadcasting a user's edits to all other users connected to the same shared note.

2.2 Front-end

2.2.1 Index page

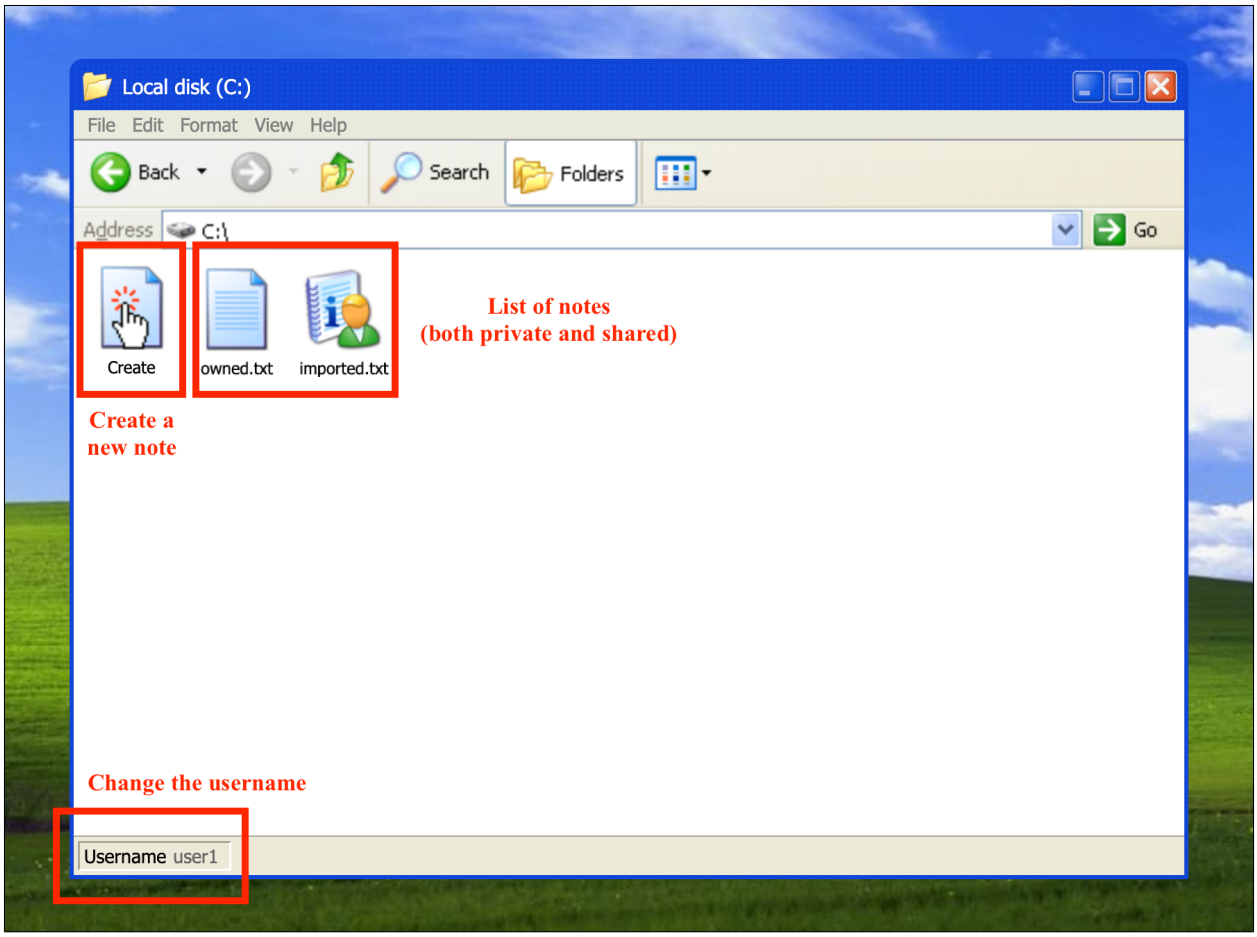

Figure 2: Index page of the front-end web application

The user interface is themed like Windows XP for no reason at all other than the fact that we liked it.

On the index page, users can set their username (located at the bottom left, randomized by default), create a new note, and view their list of notes. Notes can either be *owned* (created by the user) or *imported* (owned by another user and accessed via a shared link). The user's username and note list are persisted in the browser's **Local Storage**.

When creating a new note, the browser generates an UUID, prompts the user for a note name, and then redirects to `/note?uuid=...`. The front-end code will then parse such url, find the UUID parameter and open a WebSocket connection with the load balancer.

2.2.2 Editor page

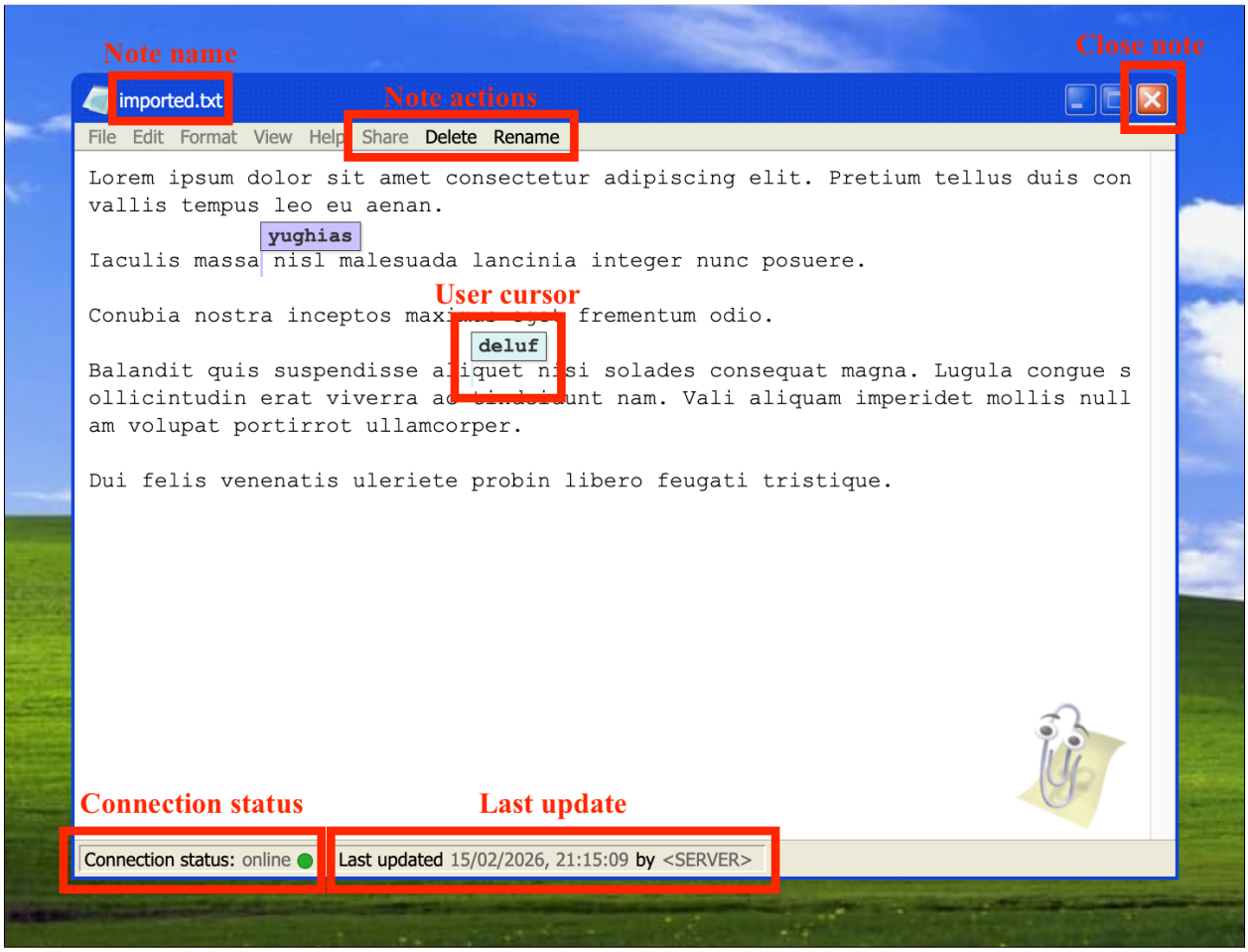

Figure 3: Editor page of the front-end web application

This is the page where users view and edit notes. The note's name acts as the window title. Below it, a menu bar allows users to share, rename, or delete the note. Note that deletions and renames are purely local; if a user renames or removes a note from their list, only they see the change.

The "Share" button generates a link in the format `/note?uuid=...&name=...`. When a user opens this link, front-end JavaScript code checks if the UUID already exists in their Local Storage, and, if not, it imports the note with the specified name.

At the bottom of the window, an indicator shows the WebSocket connection status to the server, and the timestamp of the last received update. Upon initial page load, the entire note state is fetched, making the `<SERVER>` itself the "last updater".

If the connection drops, the client attempts to reconnect using an **exponential back-off** strategy. Any **offline edits** made to the note **are stored locally** and pushed to the server upon successful reconnection.

The note's content is housed in an editable HTML `textarea`, restricted to accept only single-character edits (as anticipated in Section 1.1). Callback functions are bound to events like `input` (for text modifications) and `selectionchange` (for cursor movements) to transmit updates to the server (more on that later).

It is worth highlighting how we solved the non-trivial **problem of rendering remote users' cursors**. Suppose we receive an update that a specific user is typing a character, say X. We face two distinct UI challenges:

1. Calculating the exact x, y coordinates (relative to the `textarea`) where the remote cursor should be drawn

2. Calculating how remote cursors should visually shift (especially complex when factoring in text wrapping and newlines)

The first problem could theoretically be solved "manually" by using a monospace font with a fixed line-height. However, this is both inefficient, as it requires looping through all the characters up to X, and unreliable, as different browsers might have rendering quirks that are difficult to debug.

Instead, we adopt a more elegant strategy shown in Figure 4: we create a hidden `div` that perfectly mirrors the exact CSS styles and dimensions of the real `textarea`. We copy all text up to character X into this hidden `div` and append a dummy `span` child element. The browser's layout engine automatically positions this `span` exactly where the cursor should be. We then retrieve the precise coordinates of the `span` relative to the hidden `div`'s top-left corner and use those same coordinates to render the cursor over the real `textarea`.

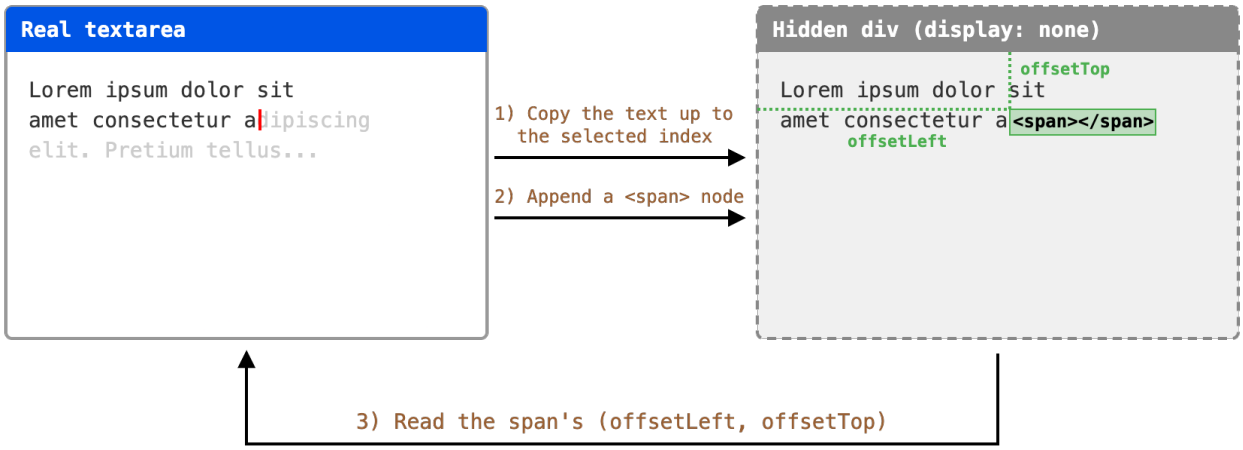

Figure 4: Computation of a cursor's coordinates relative to the `textarea`

The second problem (shifting remote cursors when local text changes) is difficult to solve locally because a single added character might trigger line-wraps that move all subsequent text. We solve this by making a slight latency trade-off: remote cursors are *not* immediately shifted during local text modifications. Instead, once our local edits propagate to the remote users, their cursors will move naturally on their end, triggering an update that is sent back to us, correcting their cursor positions on our screen.

Finally, because notes frequently exceed the visible page, a scrollbar is present. Without diving deep into it, we implemented a scrolling synchronization mechanism to ensure that remote cursors stay properly aligned with the text even when the user scrolls.

2.3 Consistency model

To understand the back-end architecture, we must first address the core technical challenge of this project: managing consistency across concurrent edits.

Given our simplifying assumption of plaintext-only data, a note can fundamentally be represented as a string of characters. Consequently, the first solution that comes to mind is to broadcast edits using absolute string indexes, for example:

```
{ "username": "user1", "action": "INSERT", "index": 45, "char": "X" }
{ "username": "user3", "action": "DELETE", "index": 37 }
{ "username": "user6", "action": "MOVE", "index": 142 }
```

Imagine the text is `"ABC"` for two clients, and they concurrently execute the following edits:

```
{ "username": "user1", "action": "INSERT", "index": 1, "char": "X" }
```

```
{ "username": "user2", "action": "DELETE", "index": 2 }
```

`user1` intends to insert `X` between `A` and `B`, while `user2` intends to delete `C`. If `user2`'s request arrives at the server first, the intent of each user is preserved and everything works fine. However, if `user1`'s request arrives first, then the string becomes `"AXBC"`. When `user2`'s request is subsequently applied at index 2, it mistakenly deletes `B` instead of `C`!

A traditional solution to this is the **Operational Transformations** (OT) theory. The core concept of OT is to transform the parameters of incoming editing operations based on the effects of previously executed concurrent operations. In our example, the server would adjust user2's deletion index from 2 to 3 to account for user1's insertion.

While OT is widely used, especially with textual updates, it is notoriously difficult to understand and implement correctly. In fact, even the foundational paper on OT[1] contained subtle flaws[2] that went undiscovered for several years[3].

However, for our use case, there is an easier solution: if, rather than relying on absolute string indices, we assigned a unique, globally ordered ID to every single character, then operations would become entirely independent of one another. This is the concept behind **Conflict-free Replicated Data Types** (CRDTs).

[1] Ellis & Gibbs (1989), *Concurrency control in groupware systems.*
[2] *A Survey on Operational Transformation Algorithms: Challenges, Issues and Achievements.*
[3] Figma Technical Blog: *Realtime editing of ordered sequences.*

Consider the same example as before, using CRDTs, the text can be represented as:

```
[{"id": 10, "char": "A"}, {"id": 20, "char": "B"}, {"id": 30, "char": "C"}]
```

The concurrent requests then become:

```
{ "username": "user1", "action": "INSERT", "id": 15, "char": "X" }
```

```
{ "username": "user2", "action": "DELETE", "id": 30 }
```

where `15` is a unique ID ordered between `A` and `B`.

Because they target absolute IDs, the arrival order no longer matters! Eventual consistency is guaranteed!

These identifiers are known as **fractional indices** because, when inserting a new character between two existing ones, you calculate a "fractional" value between their two IDs. Because floating-point numbers quickly suffer from precision loss, fractional IDs are typically implemented as strings.

Generating efficient fractional IDs, i.e., finding a *short* string ID between IDs X and Y, is a complex problem. For example, if you always take the midpoint then the string ID grows large very quickly (check this article for more info).

Moreover, to minimize the likelihood of index collisions when generating fractional indexes concurrently, random jitter can be added. We rely on the jittered-fractional-indexing open-source library to handle this math.

Using the library's default $b = 30$ bits of random jitter ($B = 2^{30}$ possible jitters), we can estimate collision probability using the Birthday Paradox. If $N = 100$ users simultaneously generate an index in the exact same gap, the probability p of a collision is:

$$p \approx 1 - \exp\left(-\frac{N^2}{2B}\right) = 4.66 \times 10^{-6}$$

This translates to a collision chance of less than 0.0005%, which is more than sufficient for our application's scale.

2.4 Back-end

2.4.1 Exchanged messages

The system relies on two primary categories of WebSocket messages (both JSON-encoded):

Edit messages As discussed, these are generic messages sent by the client to broadcast character insertions, deletions, and cursor movements.

```
{
    "action": "INSERT",
    "username": "deluf",
    "id": "ZzW30Jf",
    "char": "A"
}
```

```
{
    "action": "DELETE",
    "username": "yughias",
    "id": "ZzW30Jf"
}
```

```
{
    "action": "MOVE",
    "username": "deluf",
    "id": "ZzW30Jf"
}
```

```
{
    "action": "DISCONNECT",
    "username": "yughias"
}
```

Synchronization messages Sent by the server immediately upon a successful connection.

If the number of clients connected to the same note is greater than a certain server-defined threshold, new clients are queued. The `QUEUED` message is used to tell the client that he is being queued and his position in queue.

```
{
    "action": "QUEUED",
    "position": 4
}
```

When the queue is over (or if queuing was not necessary), clients receive a `SYNC` message, which provides the full textual state of the note, as well as the active cursor positions of all other connected clients.

```
{
    "action": "SYNC",
    "data": [
        { "id": "ZzW30Jf", "char": "A" },
        { "id": "a02zZWH", "char": "B" },
        { "id": "a0vS3OR", "char": "C" }
    ],
    "cursors": [
        { "id": "a0vS3OR", "username": "deluf" },
        { "id": "a02zZWH", "username": "yughias" }
    ]
}
```

`SYNC` messages can also be explicitly requested by clients via a `SYNCREQ`. This is useful only for a very specific scenario regarding locally-queued edits, as explained in the `flushQueuedEdits()` function in `collaborativeSocketClient.js`.

2.4.2 Erlang code structure

The project relies on external dependencies, such as: **cowboy** for HTTP and WebSocket management and **jsx** for JSON encoding and decoding. We decided to utilize **rebar3** to manage them.

The application is implemented by the following modules:

`collaborative_editor_app.erl` It's the entry point of the application, its `start/2` function is responsible for bootstrapping the entire system: it attempts to integrate the node into a larger distributed cluster by pinging IP addresses defined in the `join_nodes` environment variable, initializes Mnesia's tables (as explained in Chapter 2.4.3), compiles the Cowboy routing dispatch table, which maps the `/:doc_id` HTTP endpoint directly to the WebSocket handler, and, finally, it starts the Cowboy HTTP listener.

`ws_handler.erl` This module governs the lifecycle of individual client WebSocket connections. Upon a new connection request, the `init/2` function takes the `doc_id` parameter from the requested URL (`cowboy_req:binding/2`) add saves it to the connection state. Once the connection is upgraded to a WebSocket via `websocket_init/1`, the handler joins the user to the editing session. It then acts as a bidirectional translation layer: it decodes incoming JSON messages into Erlang maps using `jsx:decode/2`, mapping actions like `SYNCREQ`, `INSERT`, `DELETE`, and `MOVE` to internal `doc_server` casts. Conversely, its `websocket_info/2` callbacks receive internal Erlang messages (such as queue updates or concurrent peer edits) and encode them back into JSON strings to push to the client.

`doc_registry.erl` In a distributed environment, a single document might be requested by clients connected to entirely different physical nodes. To maintain a single source of truth, this module leverages Erlang's `global` module to register document servers across the entire cluster. When `get_server/1` is invoked, it checks if a process named `{doc, DocId}` already exists anywhere in the network. If not, it safely spawns one. This globally locked registration prevents inconsistent scenarios where multiple nodes inadvertently instantiate isolated erlang processes for the exact same document.

`crdt_core.erl` This module encapsulates the logic of our Conflict-free Replicated Data Type. The document is modeled as a Erlang list of `{ID, Char}` tuples. Character insertions utilize simple list prepending (`[{ID, Char} | Doc]`), while deletions use `lists:keydelete/3`.

`doc_server.erl` This is the functional core of the application, specifically, it handles:

- **State management:** the server's state record (`#state{}`) holds the document ID, the CRDT document list, a map of active cursors, a list of active process IDs (`pid`s), a waiting queue of `pid`s, a mapping of `pid`s to usernames, and an operation counter
- **Fault detection:** when a user connects, the server explicitly monitors its WebSocket process using `erlang:monitor/2`. If a client's connection drops unexpectedly, the Erlang VM automatically sends a `'DOWN'` message to the `doc_server`, which handles this by gracefully removing the user's cursor, broadcasting a `DISCONNECT` event to remaining peers, and automatically promoting the next user in the queue to an active editing state. If no users are left, the document is saved on Mnesia and the Erlang process relative to that document is shutdown. **It is worth noting that monitoring is bidirectional: each WebSocket process also monitors its `doc_server` process**
- **Concurrency control:** to prevent system overload, it limits concurrent editors to `?MAX_ACTIVE`. If additional users attempt to join, their `pid` is appended to a `queue` list, and they receive `queue_update` messages containing their position in queue

2.4.3 Mnesia

Initialization Unlike traditional monolithic databases (like MySQL), Mnesia is embedded dynamically during the application's startup routine via the `init_mnesia/0` function. The

system executes a the following sequence to ensure data integrity across the decentralized cluster:

1. The local node explicitly alters the base Mnesia `schema` copy type to `disc_copies`. This step ensures that the structural definition of the database is persisted to the local physical disk, surviving system reboots.

2. It then attempts to create the core `editor_docs` table, configured to hold `doc_id` and `content` attributes, with its local storage type also set to `disc_copies`.

3. In a distributed setting, another node might have already created this table. If `mnesia:create_table/2` returns an `{already_exists, editor_docs}` error, the local node catches this correctly. It subsequently calls `mnesia:add_table_copy/3`, effectively requesting a full, peer-to-peer synchronization of the existing data from the remote nodes to its local disk.

4. To guarantee safety, the application uses `mnesia:wait_for_tables/2` to block further execution until the database is fully loaded and synchronized into memory.

Intelligent write-back caching Collaborative text editing presents a unique database challenge: users generate a continuous, massive stream of granular, single-character operations. Executing a dedicated database transaction for every single keystroke would saturate disk I/O and introduce unacceptable latency.

To circumvent this, the `doc_server` process functions as an intelligent write-back cache. We decided to write to Mnesia whenever at least one of the following three conditions is true:

1. **Operation volume threshold:** The server maintains an internal `op_count` integer. After every `?SAVE_EVERY` distinct operations are accumulated and applied to the in-memory CRDT, the entire state is flushed to Mnesia

2. **Temporal interval:** the server initializes a repeating timer upon startup. Every `SAVE_INTERVAL` milliseconds, a `trigger_save` message is handled. If `op_count > 0` (meaning new edits have occurred since the last flush), the state is written to the database and the counter resets

3. **Termination of the sharing a document:** When the `erlang:monitor` detects that the last active user has disconnected, the `doc_server` ends. Before returning `{stop, normal, State}`, it performs one final, definitive `dirty_write` to ensure the latest document state is safely archived

Furthermore, because the `gen_server` strictly serializes all incoming operations for a given document, locking mechanisms are redundant. Therefore, when writing back cached edits to the database, the server utilizes `mnesia:dirty_write/1` rather than traditional ACID transactions, heavily maximizing throughput while relying on the Erlang process itself to

maintain consistency. When a document is later reopened, the newly spawned `doc_server` immediately executes a `mnesia:dirty_read(editor_docs, DocId)` to rapidly fetch the archived `#editor_docs{}` record and load the `content` back into active memory.

2.4.4 Deployment

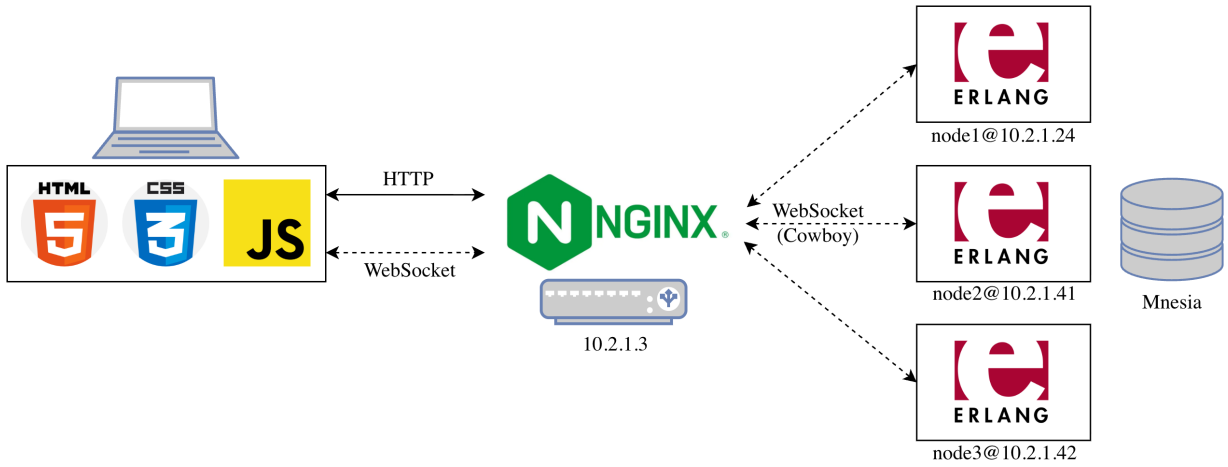

Figure 5: Deployment diagram

The project is deployed on a cluster composed by four virtual machines of the datacenter of the University of Pisa: one VM is dedicated to running the NginX web server and load balancer, while the other three are used to run the Erlang back-end.

Specifically, to start an Erlang node we run the following command:

```
rebar3 shell --name node{N}@{IP} --setcookie collaborative-text-editor --config
↪  node.config
```

where `node.config` is:

```
1 [{
2     collaborative_editor, [
3         {http_port, 8086},
4         {join_nodes, ['node1@10.2.1.24', 'node2@10.2.1.41', 'node3@10.2.1.42']}
5     ]
6 }].
```

Finally, the NginX configuration (`/etc/nginx/sites-available/default`) is:

```
# ------------------------------------------------------
#  Global settings
# ------------------------------------------------------

upstream erlang_nodes {
    # Always map the same URLs (i.e., the same notes) to
    #  the same servers (unless they are down)
    hash $uri consistent;
    server 10.2.1.24:8086;
    server 10.2.1.41:8086;
    server 10.2.1.42:8086;
}

# ------------------------------------------------------
#  Web server for static files [port 80]
# ------------------------------------------------------
server {
    listen 80;

    # Where the server root (/) points to
    root /var/www/html;

    # Accept requests for any domain name or IP address that
    #  ends up hitting this port
    server_name _;

    # Default map: for every request, try an exact file match,
    #  then an exact folder match, finally 404.
    # This is required to serve the index.html page (/) and
    #  static assets like /css/*.css, /js/*.js, /media/*.png
    location / {
        try_files $uri $uri/ =404;
    }

    # Map /note path to note.html file.
    # Without this, users would have to request exactly note.html
    location = /note {
        try_files /note.html =404;
    }
}
```

```
# ------------------------------------------------------
#  WebSocket load balancer [port 8080]
# ------------------------------------------------------
server {
    listen 8080;

    # Accept requests for any domain name or IP address that
    #  ends up hitting this port
    server_name _;

    location / {
        # Forward the request to the server pool 'erlang_nodes'
        proxy_pass http://erlang_nodes;

        # WebSocket connections actually start as plain HTTP connections
        #  that are then 'upgraded' to WebSockets via the HTTP Upgrade:
        #  header (accessible via the $http_upgrade variable).

        # The HTTP-to-WebSocket upgrade process requires HTTP version
        #  1.1, but NGINX defaults to HTTP 1.0 when forwarding to
        #  backend servers
        proxy_http_version 1.1;

        # We now need to adjust a few headers that are by default
        #  stripped or wrong when performing request forwarding

        # Pass the exact Upgrade: HTTP header the client sent
        proxy_set_header Upgrade $http_upgrade;

        # Pass the exact Connection: HTTP header the client sent
        proxy_set_header Connection $http_connection;

        # Pass the original domain name/ip the client requested.
        # Otherwise would send http://erlang_nodes/...
        proxy_set_header Host $host;

        # Pass the IP of the client, not the one of the load balancer
        proxy_set_header X-Real-IP $remote_addr;

        # Prevent NGINX from dropping long-lived, idle WebSocket connections.
        # This overrides the default timeout of 60 seconds.
        # There is a more restrictive timeout on Erlang CowBoy anyway
        proxy_read_timeout 3600s;
        proxy_send_timeout 3600s;
    }
}
```

3 Testing

Sudden client disconnection In this scenario, two users are concurrently typing in the same document when one user's connection is forcefully dropped. Because `doc_server` explicitly monitors each connected WebSocket process (`erlang:monitor/2`), it immediately receives a `'DOWN'` message on WebSocket crash. The `doc_server` removes the disconnected user from the `active` list, broadcasts a `remove_cursor` event to the remaining peers, and automatically promotes the first waiting process from the `queue` to an active state. When the disconnected client subsequently recovers and reconnects, the WebSocket handler takes the `doc_id` from the URL bindings to rejoin the same session as before. However, if the `?MAX_ACTIVE` threshold is currently saturated by other users, the returning user is placed back into the queue.

Idle user disconnection If a user remains completely inactive for an extended period, the WebSocket connection automatically times out and closes. The backend handles this identically to a sudden crash (via the `'DOWN'` message). This mechanism is crucial for preventing resource starvation; it ensures that users who go AFK (away from keyboard) on purpose cannot permanently occupy active editing slots while others are queued.

Erlang node crash This test simulates a catastrophic failure where a whole Erlang node suddenly dies. Client-side, the WebSocket closes unexpectedly. The clients' exponential back-off strategy kicks in, attempting to open a new connection to the load balancer. The Nginx load balancer seamlessly routes this new traffic to the remaining healthy nodes in the cluster. Upon client connection, the selected node's `doc_registry` detects that the server for that `doc_id` is no longer globally registered and spawns a fresh `doc_server`, which reads the latest state from the Mnesia database. However, because of our write-back mechanism, at most `?SAVE_EVERY` operations prior to the crash may have been permanently lost.

Unexpected termination of `doc_server` process In this scenario, two users were actively editing the same document, when all of a sudden the `doc_server` process is manually killed from the Erlang shell. Since each `ws_handler` process also maintains a monitor on the corresponding `doc_server`, as soon as the server terminates, the handler immediately receives a `'DOWN'` message and responds by stopping itself, which in turn closes the WebSocket connection. On the client side, this closure is detected and triggers the exponential back-off logic. Upon reconnection, the request is routed through Nginx to a working node. The document state is restored from the Mnesia database, ensuring consistency across clients. As in the previous scenario, due to the write-back caching strategy, recent operations may got lost.

Sustained concurrent usage To evaluate performance limits under heavy concurrent use, we benchmarked a single document session with the queuing mechanism disabled. We simulated 5, 10 and 15 users typing continuously at 600 edits per minute (on average, a word

consists of 6 characters) (a fast typing speed), splitted in 80% inserts, 10% deletes, 10% moves, and we recorded the response times of the `SYNCREQ` message at regular intervals.

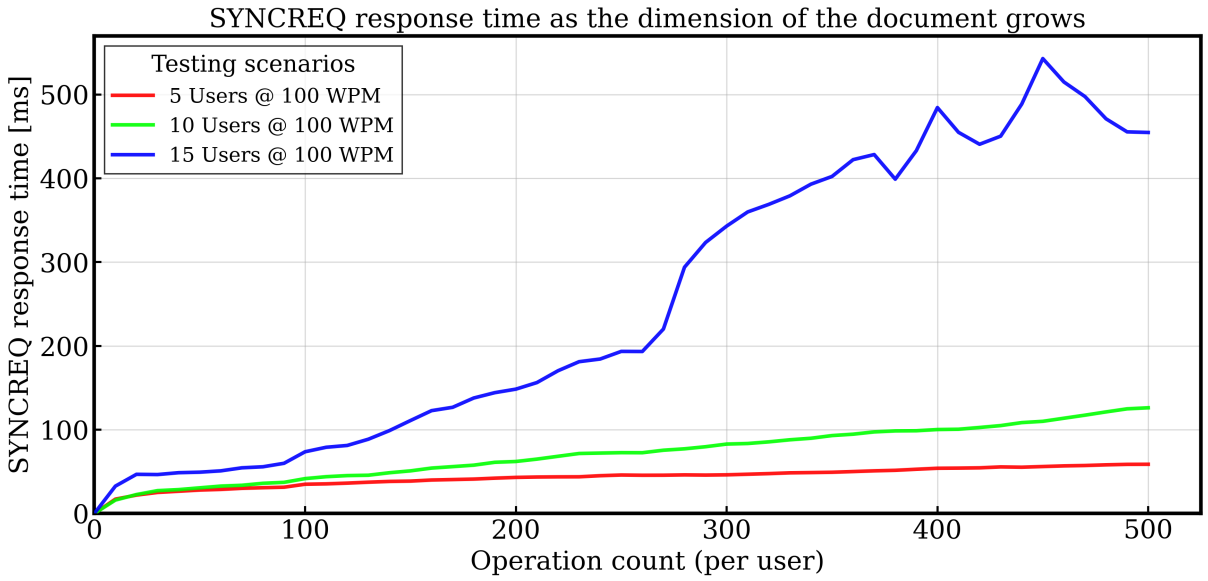

Figure 6: `SYNCREQ` response times as the dimension of the document grows

The results, illustrated in Figure 6, reveal that:

- With 5 users, the response times stay relatively constant. A small, linear, growth is expected, since the server replies to a SYNCREQ with the full state of the document, which grows over time.
- With 10 users, the response times are still linear, with a slope that looks exactly double in magnitude. This is once again expected, and well within reasonable performance thresholds
- With 15 users instead, the response times are not linear anymore. This degradation occurs because the effect of every operation must be broadcast to all clients, rapidly filling the processes' mailboxes.

Establishing a maximum of 10 active concurrent users provides an optimal threshold, balancing necessary collaborative parallelism with system stability to maintain real-time synchronization.

Database write frequency To determine the optimal frequency for saving document state to disk, we evaluated the processing time for 10'000 sequential WebSocket insert operations across varying `?SAVE_EVERY` intervals. This parameter indicates how many operations are held in the process state before triggering a write to the Mnesia database.

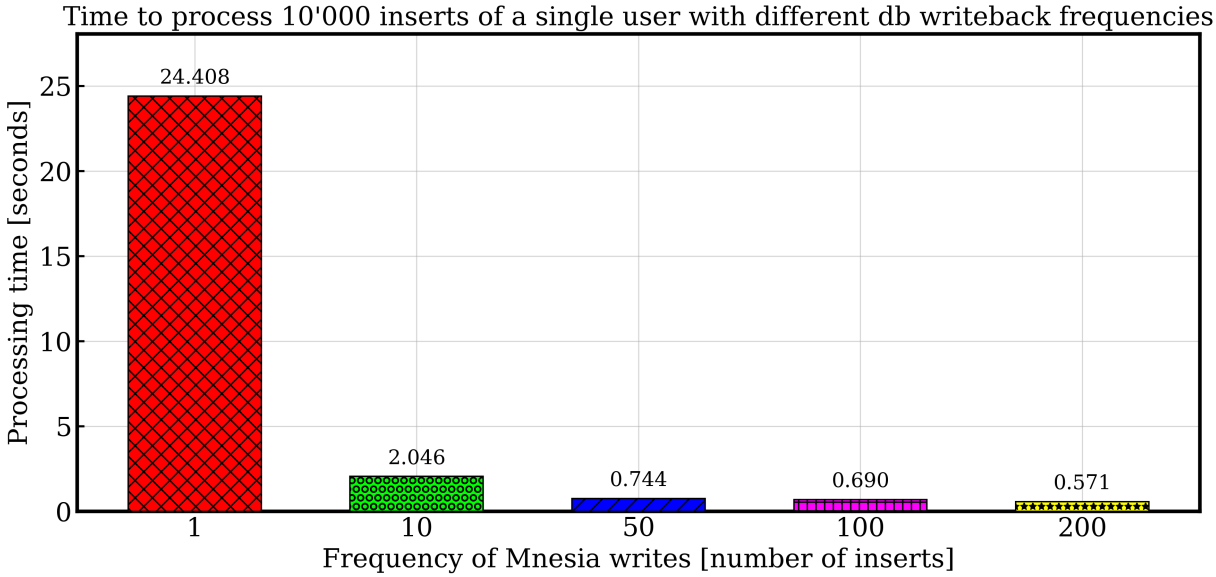

Figure 7: Time to process 10'000 inserts of a single user with different db writeback frequencies

The benchmark reveals that, as expected, there is a severe I/O bottleneck when writing to disk on every single operation (`save_every = 1`), resulting in a total processing time of 24.408 seconds. Batching the disk writes yields immediate performance improvements. At an interval of 10 operations, processing time drops by over 90% to 2.046 seconds.

We ultimately selected `save_every = 50` as the optimal configuration threshold. At this interval, the total processing time falls to 0.744 seconds, effectively reducing the database latency bottleneck. While larger intervals (100 and 200) offer slight additional speed improvements, these gains represent steeply diminishing returns. Furthermore, higher intervals dangerously expand the volatility window; in the event of a node crash, up to 199 un-synced operations could be lost. An interval of 50 strikes the ideal architectural balance, ensuring real-time processing while strictly limiting the potential for data loss.