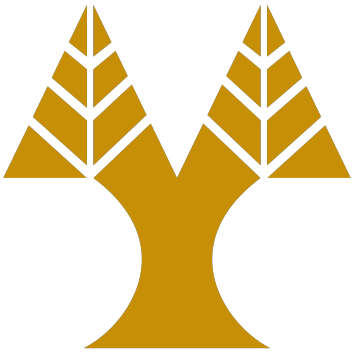
Diploma Project

COLLABORATIVE TEXT-EDITING OVER DISTRIBUTED SHARED MEMORY

Michalis Christodoulou

UNIVERSITY OF CYPRUS



DEPARTMENT OF COMPUTER SCIENCE

May 2026

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

COLLABORATIVE TEXT-EDITING OVER DISTRIBUTED SHARED MEMORY

Michalis Christodoulou

Advisor: Professor Chryssis Georgiou

The Thesis was submitted in partial fulfillment of the requirements for obtaining the Computer Science degree of the Department of Computer Science of the University of Cyprus

May 2026

I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and with full accountability on my part for the accuracy, integrity, and validity of all content.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Chryssis Georgiou, for his guidance, support, and valuable feedback throughout the course of this thesis.

I would also like to thank Algolysis Ltd, and in particular Nicolas Nicolaou and Andria Trigeorgi, for their support and collaboration during this work. The regular meetings and discussions with my supervisor and the Algolysis team were extremely valuable in shaping the architecture, design decisions, implementation, and evaluation of the system.

Finally, I would like to thank my family for their continuous encouragement and support throughout my studies, as well as my fellow students and friends for their help, motivation, and companionship during this journey.

ABSTRACT

Real-time collaborative editors typically rely on centralized synchronization servers, commonly implemented with WebSockets, to distribute document updates between clients. This thesis investigates an alternative architecture in which collaborative editing is supported through a Distributed Shared Memory (DSM) backend instead of a centralized server. The system implements a web-based text editor using Yjs, a Conflict-free Replicated Data Type (CRDT) framework, and Quill, a rich-text editor, allowing multiple clients to edit the same document concurrently. The proposed design stores document state in a coverable DSM through HTTP-based read and write operations. To support concurrent updates, clients periodically read the latest document state from the DSM, merge it with their local CRDT state, and write back the merged result only when there are unpersisted local edits, retrying when the DSM's coverability guarantee rejects a conflicting write. Because document state is held by the replicated DSM rather than a single server process, the architecture removes the dedicated CRDT-aware synchronization server as a single point of failure, replacing it with a replicated DSM backend while preserving consistency across distributed clients. The implementation was evaluated through Selenium-based multi-client experiments with 1 to 16 concurrent clients. The results demonstrate correct convergence in all tested configurations, with no missing, duplicated, or extra generated tokens, while also revealing the expected contention cost of the shared-snapshot design. A comparison with a conventional `y-websocket` system highlights the main trade-off: the `y-websocket` system achieves lower propagation latency and faster convergence, while the DSM-based system preserves correctness using a generic replicated storage backend rather than a dedicated Yjs-aware synchronization server.

Keywords: Distributed Shared Memory, Yjs, CRDT, real-time collaboration

TABLE OF CONTENTS

ABSTRACT iv

TABLE OF CONTENTS v

LIST OF TABLES viii

LIST OF FIGURES ix

LIST OF ABBREVIATIONS x

1 Introduction 1

1.1 Motivation and Purpose of the Thesis . . . 1
1.2 Methodology . . . 2
1.3 Contributions . . . 3
1.4 Document Organization . . . 4

2 Background and Related Work 6

2.1 Collaborative Editing and Consistency Maintenance . . . 6
2.1.1 The Collaborative Editing Problem . . . 6
2.1.2 Operational Transformation . . . 7
2.1.3 Conflict-free Replicated Data Types . . . 7
2.2 YATA and Yjs . . . 8
2.2.1 The YATA algorithm . . . 8
2.2.2 Yjs Document Updates and State Vectors . . . 9
2.2.3 Use of Yjs Primitives in the Prototype . . . 9
2.2.4 Why Yjs Fits the Proposed Architecture . . . 10
2.3 Quill Rich-Text Editor . . . 10
2.4 Distributed Shared Memory . . . 11
2.4.1 Atomic Distributed Shared Memory . . . 11
2.4.2 The ABD algorithm . . . 11
2.4.3 ARES and Reconfigurable Atomic Storage . . . 12

2.4.4 Coverability and Coverable ARES 13

3 Design and Implementation **15**
3.1 Conventional Yjs Architecture and Design Objective 15
3.1.1 Conventional Yjs Client-server Architecture 15
3.1.2 Conventional Update Propagation 16
3.1.3 Design Objective of the Proposed System 16
3.2 DSM Interface Available to the Prototype 16
3.3 Design Evolution 17
3.3.1 Initial Per-update Approach 17
3.3.2 Hybrid Snapshot-plus-updates Scheme 18
3.4 Final System Architecture and Document Storage Model 19
3.4.1 Final DSM-based Architecture 19
3.4.2 Document Storage Model 20
3.5 Snapshot-based Synchronization Algorithm 22
3.5.1 Algorithm Overview 22
3.5.2 Polling Loop: Reading the Latest DSM Snapshot 22
3.5.3 Sync Loop: Applying Remote State and Flushing Local Changes . . . 24
3.5.4 Detecting Local Unwritten Changes 27
3.5.5 Conflict Handling and Coverability Retries 28

4 Demonstration of the Collaborative Editing Prototype **30**
4.1 Prototype Interface Overview 30
4.2 Starting a Collaborative Session 31
4.3 Runtime Diagnostics and Synchronization Observability 34
4.4 Collaborative Editing Across Multiple Clients 35
4.4.1 Multi-client Setup 35
4.4.2 Initialization of the Shared DSM Object 35
4.4.3 Writing and Observing Shared Document State 36
4.4.4 Concurrent Editing and Retry Behavior 37
4.4.5 Joining or Rejoining an Existing Session 38

5 Experimental Evaluation **39**
5.1 Evaluation Objectives 39
5.2 Experimental Setup 39
5.3 Benchmark Workload 40
5.4 Measured Metrics 40
5.5 Correctness Validation 41

5.6 Results and Discussion . . . 41
5.6.1 Correctness Results . . . 41
5.6.2 Aggregate Throughput vs Concurrent Clients . . . 41
5.6.3 Write Outcome Composition vs Concurrent Clients . . . 42
5.6.4 Read Latency During the Active Writing Phase . . . 43
5.6.5 Write Latency During the Active Writing Phase . . . 44
5.6.6 Read Latency Percentiles vs Concurrent Clients . . . 45
5.6.7 Write Latency Percentiles vs Concurrent Clients . . . 46
5.7 Comparison with the Conventional `y-websocket` Architecture . . . 47
5.7.1 Comparison Setup and Propagation Latency Metric . . . 47
5.7.2 Propagation Latency Comparison . . . 48
5.7.3 Convergence Time Comparison . . . 49
5.8 Limitations . . . 50

6 Conclusion and Future Work **52**
6.1 Conclusion . . . 52
6.2 Limitations . . . 53
6.3 Future Work . . . 54

BIBLIOGRAPHY **55**

LIST OF TABLES

3.1 DSM operations relevant to the prototype. 17

5.1 DSM stress-test benchmark configuration. 40

5.2 End-to-end propagation latency for the DSM-based system and the conventional `y-websocket` system. Values are in milliseconds and represent the mean over three runs. 49

LIST OF FIGURES

3.1 Conventional Yjs client-server architecture. 15
3.2 Propagation of a document update through a conventional Yjs WebSocket server. 16
3.3 Final DSM-based collaborative editing architecture. 21
3.4 Client initialization and startup of the snapshot synchronization loops. 23
3.5 Polling loop for reading the current DSM snapshot. 24
3.6 Two-phase sync loop for applying remote state and publishing local changes. . 26

4.1 Main interface of the DSM-based collaborative editing prototype 31
4.2 Collaborative session configuration before initialization 32
4.3 Coverable DSM Node selection dropdown menu 32
4.4 Collaborative editing session after successful initialization 33
4.5 Runtime diagnostics panel of the collaborative editing prototype 34
4.6 Two clients initialized on the same document identifier through different DSM nodes 35
4.7 Client A inserting text into the shared editor 36
4.8 Client B observing the newer shared document state 37
4.9 Concurrent text insertion from two clients, showing successful writes, retries, and DSM tag progression 38

5.1 Aggregate read and write throughput as the number of concurrent clients increases. 42
5.2 Write outcome composition as the number of concurrent clients increases. . . . 43
5.3 Read latency during the active writing phase. 44
5.4 Write latency during the active writing phase. 45
5.5 Read latency percentiles as the number of concurrent clients increases. 46
5.6 Write latency percentiles as the number of concurrent clients increases. 47
5.7 Time from bot stop until all clients share the same final text. Error bars show ± 1 standard deviation over three runs. 50

LIST OF ABBREVIATIONS

DSM	Distributed Shared Memory
OT	Operational Transformation
CRDT	Conflict-free Replicated Data Type

Chapter 1

Introduction

1.1	Motivation and Purpose of the Thesis	1
1.2	Methodology	2
1.3	Contributions	3
1.4	Document Organization	4

1.1 Motivation and Purpose of the Thesis

Real-time collaboration has become an essential feature of modern productivity software [5]. Tools such as Google Docs, Microsoft 365 and Etherpad let many users edit the same document concurrently, with each participant's changes appearing on every other participant's screen within fractions of a second. Beyond convenience, this style of collaboration has practical importance: it eliminates the "send file, wait, merge by hand" cycle that used to dominate joint authorship, replacing it with a shared workspace whose state is consistent across all participants.

The conventional architecture for such systems can be understood as combining two concerns: document-level conflict resolution and a communication layer for exchanging changes between replicas [18]. The conflict-resolution mechanism allows multiple clients to edit the same document concurrently while still ensuring that replicas converge to a consistent state. Two families of techniques are used in practice. Operational Transformation (OT) [3, 16, 17], the older approach, used by systems such as Google Docs and Etherpad, transforms each incoming operation against the concurrent operations already applied so that all replicas converge to the same state. Conflict-free Replicated Data Types (CRDTs) [1, 12, 15], by contrast, represent the document with data structures designed so that replicas receiving the same set of updates converge to the same state, without requiring server-side transformation. The work in this thesis is built on a CRDT library, specifically Yjs [20].

The second concern is the communication layer that propagates document updates between connected clients. In many conventional collaborative editing systems, this role is played by a dedicated synchronization server, often using WebSockets. The Yjs ecosystem, for example, pro-

vides the `y-websocket` provider [22], which implements a conventional client-server model in which clients connect to a single WebSocket endpoint and the server distributes document updates among them. This pattern is convenient for application developers, but it has two structural drawbacks. First, in its standard single-server deployment, it introduces a single point of failure for every collaboration session it serves, since the server process acts as the central synchronization point for connected clients. Second, the synchronization server is tightly coupled to the Yjs layer: rather than acting as a generic storage service, it handles Yjs document updates, decodes the binary synchronization messages defined by the Yjs protocol, and participates in the protocol-specific handshake used to reconcile replicas [22, 23].

This thesis explores an alternative architecture that addresses both drawbacks at once. Instead of a dedicated, CRDT-aware synchronization server, the proposed system uses a generic DSM service, specifically the coverable DSM [4, 11], as the only backend.

The DSM exposes a small HTTP API of read and write operations on opaque key-value pairs, replicates each value across multiple nodes internally, and rejects any write whose base value is no longer the most recent. The DSM is independent of the Yjs data model: it does not decode Yjs updates, maintain a Yjs document, execute the Yjs synchronization protocol [23], or perform CRDT-specific merge logic. Instead, it stores the encoded document state as an opaque payload, while all Yjs-specific processing is performed by the browser-side clients.

The questions investigated in this thesis are whether such an architecture preserves correctness under concurrent editing, what its performance characteristics are as concurrency grows, and what the cost of its synchronization algorithm over the DSM is in practice.

1.2 Methodology

The methodology of this thesis combines system design and implementation with experimental evaluation.

A web-based prototype editor was implemented in JavaScript, combining the Yjs CRDT framework, the Quill rich-text editor [13], and a custom synchronization layer that uses the coverable DSM as its only backend. The synchronization layer runs entirely in the browser, while the coverable DSM itself is treated as an external service consumed through its HTTP API. The final synchronization algorithm was reached through iterative development. Earlier approaches based on per-update key storage and a hybrid snapshot-plus-updates scheme were abandoned as their shortcomings became apparent, leading to the snapshot-only design presented in this thesis.

The prototype was evaluated through a multi-client benchmark script written in Python. The script uses Selenium WebDriver [14] to drive several Chrome browser instances on the same

host, each loaded with the prototype editor and connected to one of four DSM nodes in a round-robin fashion. Inside every browser, an automated bot continuously inserts uniquely numbered tokens into the editor at a fixed rate, while the browser-side synchronization layer records throughput, per-operation latency, and write outcome statistics, including successful writes and retries caused by coverability conflicts. After a fixed measurement window the script collects the metrics from each client, waits for the editors to converge, and verifies that all clients converge to the same document text and that every generated token appears exactly once. The benchmark was run at 1, 2, 4, 8, and 16 concurrent clients, and each concurrency level was repeated three independent times, so that every reported data point is the mean of three runs, with standard deviation shown on the relevant charts. In addition to the DSM-specific benchmark, a second experiment was conducted to compare the proposed architecture with a conventional `y-websocket` system. Since DSM read/write throughput and retry statistics are specific to the DSM-based synchronization algorithm, the comparison used metrics that apply to both architectures: end-to-end propagation latency and convergence time. The same token-based workload and Selenium-based benchmark structure were used, allowing the DSM-based prototype and the `y-websocket` system to be compared under similar experimental conditions.

1.3 Contributions

This thesis makes five main contributions. First, it proposes an architecture for using a coverable DSM backend as the shared storage layer for collaborative editing, while keeping the CRDT-specific logic entirely on the client side. Second, it implements this architecture as a working browser-based prototype using Yjs and Quill. Third, it develops an experimental methodology for evaluating the correctness and performance of the system under concurrent editing workloads. Fourth, it compares the proposed DSM-based design with a conventional `y-websocket` architecture. Finally, it reports empirical findings on correctness, write contention, propagation latency, and convergence time as concurrency increases.

1. **Architecture and synchronization algorithm**. A design for collaborative editing over a generic DSM backend, in which all CRDT logic resides in the client and the DSM remains an opaque, replicated key-value store with no awareness of the editor's data model. The design is realised by a snapshot-based synchronization algorithm in which each document is stored under a single DSM key as a snapshot of the full Yjs document state; clients flush their local edits by re-merging with the latest snapshot and retrying when the coverability guarantee detects a conflicting write.
2. **Prototype implementation**. A working web editor combining Yjs, Quill, and the coverable DSM, with the entire synchronization layer running in the browser-side client.
3. **Evaluation methodology**. A multi-client Selenium benchmark script that measures through-

put, latency percentiles, retry rate, convergence time, and correctness across 1, 2, 4, 8, and 16 concurrent clients, with each configuration evaluated through three independent runs.

4. **Comparative evaluation**. A comparison between the proposed DSM-based architecture and a conventional `y-websocket` architecture, using end-to-end propagation latency and convergence time as shared comparison metrics.
5. **Empirical findings**. Evidence that the proposed architecture preserves correctness in every tested configuration, together with an evaluation of how the system's performance, write contention, propagation latency, and convergence time scale with concurrency.

1.4 Document Organization

The remainder of this thesis is organized as follows.

Chapter 2 provides the background required to understand the design of the proposed system and its trade-offs. It introduces the collaborative editing problem and the main approaches used for consistency maintenance, including Operational Transformation and Conflict-free Replicated Data Types. It also presents the YATA algorithm and the Yjs library, the Quill rich-text editor, and the Distributed Shared Memory concepts and algorithms relevant to the backend, including ABD, ARES, and the coverable variant of ARES.

Chapter 3 presents the design and implementation of the proposed system. It first discusses the conventional Yjs client-server architecture and the objective of replacing the conventional synchronization backend with a DSM-based approach. It then describes the DSM interface available to the prototype, the evolution of the design, the final snapshot-based storage model, and the synchronization algorithm used by the clients. The chapter also explains how the system detects local unwritten changes and handles coverability-related write retries.

Chapter 4 demonstrates the collaborative editing prototype from the user's perspective. It presents the prototype interface, the process of starting a collaborative session, and the runtime diagnostics panel used to observe synchronization behavior during execution. It also demonstrates collaborative editing across multiple clients, including multi-client initialization, shared DSM object creation, writing and observing shared document state, concurrent editing, retry behavior, and joining or rejoining an existing session.

Chapter 5 evaluates the implemented system experimentally. It first presents the evaluation objectives, experimental setup, benchmark workload, measured metrics, and correctness validation method. It then discusses the results of the DSM-based prototype, including correctness, aggregate throughput, write outcome composition, read and write latency over time, and latency percentiles as the number of concurrent clients increases. The chapter also compares the DSM-based prototype with a conventional `y-websocket` system using end-to-end update propagation

latency and convergence time. Finally, it discusses the main limitations of the evaluation, particularly the use of a single benchmark machine for all Selenium-controlled browser clients.

Chapter 6 concludes the thesis. It summarizes the main findings of the work, discusses the limitations of the current evaluation, and outlines directions for future work, including distributed benchmarking, reduced write contention, more realistic editing workloads, larger documents, late-joining clients, and further evaluation of the DSM-backed synchronization approach.

Chapter 2

Background and Related Work

2.1 Collaborative Editing and Consistency Maintenance 6

2.2 YATA and Yjs 8

2.3 Quill Rich-Text Editor 10

2.4 Distributed Shared Memory 11

2.1 Collaborative Editing and Consistency Maintenance

2.1.1 The Collaborative Editing Problem

Collaborative editing systems typically maintain replicated copies of the shared document across participating sites, allowing multiple users to modify the document concurrently over a network [3, 17]. A replica is the local copy of the shared document state maintained by each client. Users edit their local replicas independently, and the role of the synchronization mechanism is to ensure that these replicas eventually converge to the same document state.

Without a consistency maintenance mechanism, concurrent updates can cause replicas to diverge. One replica may apply operation A before operation B, while another replica may apply B before A, leading to different document states. Collaborative editing algorithms therefore aim to ensure that all replicas eventually reach the same state after receiving the same set of updates. Classical work on collaborative editing often discusses correctness in terms of convergence, causality preservation, and intention preservation [3, 16, 17]. Convergence means that all replicas become identical once all generated operations have been delivered and processed. Causality preservation means that operations that depend on earlier operations are applied in a compatible order. Intention preservation means that the effect of an operation should remain consistent with the user's original editing intention, even when it is combined with concurrent operations.

The thesis relies on an existing Conflict-free Replicated Data Type implementation, Yjs [20], for document-level convergence, and investigates whether a different backend architecture can

be used to store and exchange document state.

2.1.2 Operational Transformation

OT is one of the earliest and most influential approaches for maintaining consistency in real-time collaborative editors [16]. It was introduced for replicated group-editing systems in which users are allowed to edit their local copies immediately, without first acquiring locks, and where operations generated concurrently at different sites must later be reconciled. The central idea of OT is that an operation created on one replica may need to be transformed before it is applied on another replica, because the document state may have changed in the meantime due to concurrent operations generated elsewhere [3, 17].

A simple example illustrates the problem. Suppose two users are editing the text "$alpha\ gamma\ epsilon$". Alice inserts $beta$ after $alpha$, producing the operation *insert "beta " at offset 6*. At the same time, Bob inserts $delta$ after $alpha\ gamma$, producing the operation *insert "delta " at offset 12*. If Bob's operation is applied to the original text, it expresses his intention correctly. However, if Alice's operation reaches the synchronizer first, the document becomes $alpha\ beta\ gamma\ epsilon$. Applying Bob's original operation unchanged at offset 12 would now place $delta$ inside the word $gamma$, because Alice's earlier insertion has shifted the later characters to the right. OT resolves this by transforming Bob's operation against Alice's concurrent operation: since Alice inserted five characters before Bob's intended position, Bob's insertion offset is adjusted from 12 to 17. The transformed operation then produces the intended result, $alpha\ beta\ gamma\ delta\ epsilon$.

This example captures the essential role of OT: concurrent operations are not merged directly, but transformed in light of one another so that their intended effect is preserved. In practice, this requires the synchronization server to understand editor operations, retain enough context about concurrent edits, and apply transformation logic according to a consistent ordering policy [16, 17]. The synchronizer therefore participates directly in conflict resolution rather than acting as a passive storage service. This is important for the present thesis, which instead requires a backend that can store opaque document state without interpreting document-specific operations.

2.1.3 Conflict-free Replicated Data Types

CRDTs provide a different approach to consistency maintenance. Rather than transforming operations after conflicts arise, CRDTs are designed so that replicas can be updated independently and still converge deterministically once they have received the same set of updates [12, 15]. For collaborative text editing, this is achieved by representing the document with additional metadata, such as identifiers associated with inserted elements, instead of relying only on absolute character offsets. Concurrent updates can then be applied to the document according to

deterministic rules defined by the data type itself.

The contrast with OT can be seen using the same example. Suppose the document initially contains the sequence $alpha\ gamma\ epsilon$. Instead of describing Bob's insertion only as *insert "delta " at offset 12*, a sequence CRDT records edits relative to stable identifiers in the replicated structure. Alice may insert $beta$ between $alpha$ and $gamma$, while Bob concurrently inserts $delta$ between $gamma$ and $epsilon$. Because Bob's operation refers to the surrounding elements rather than to a position that becomes stale after Alice's edit, Alice's insertion does not invalidate Bob's intended location. Once both replicas have received both updates, they independently compute the same final sequence, $alpha\ beta\ gamma\ delta\ epsilon$. If two users insert at the same logical position, the CRDT's ordering rules provide a deterministic way to place both insertions, ensuring convergence without requiring one operation to be transformed against the other [1, 12].

CRDTs do not eliminate the need for communication between replicas, and many CRDT-based systems still use servers to relay updates. Their key architectural difference from OT is narrower: convergence does not depend on a server transforming concurrent operations. The merge semantics are provided by the replicated data type itself, which makes it possible to design systems in which conflict-resolution logic remains at the clients while the backend stores only opaque encoded state. This property is important for the present thesis, which uses a generic DSM backend rather than a dedicated CRDT-aware synchronization server.

2.2 YATA and Yjs

2.2.1 The YATA algorithm

YATA is the CRDT approach on which Yjs is based. It was proposed for near real-time peer-to-peer shared editing and was designed to support convergence, intention preservation, offline editing, and collaboration over arbitrary shared data types in Web environments [9]. YATA represents linear data, such as text, through an item-based linked structure. Insertions are integrated relative to neighbouring items rather than by absolute document offsets, since numerical positions may become unstable when several users edit concurrently. When concurrent insertions compete for the same logical position, YATA applies deterministic ordering rules so that all replicas integrate them in the same order [9].

Yjs implements an adaptation of YATA with additional implementation optimizations. Internally, Yjs is fundamentally a list CRDT: text is represented as a list of characters, although adjacent characters may be grouped into a single internal item for efficiency. Each inserted item receives a unique identifier of the form `(clientID, clock)`, where `clientID` identifies the creating replica and `clock` increases with that replica's insertions. Each item also stores refer-

ences to neighbouring items that provide the context in which it was inserted [24]. For example, if two users concurrently insert different characters between the same two existing characters, both insertions refer to the same neighbouring items but have distinct identifiers. Even if replicas receive those insertions in different orders, YATA's deterministic integration rules ensure that they are placed in the same final order at every replica.

These details are important at a conceptual level because they explain why Yjs can integrate concurrent edits deterministically. This thesis uses Yjs as an existing convergence mechanism and focuses on how its encoded state can be synchronized through a different backend architecture.

2.2.2 Yjs Document Updates and State Vectors

A collaborative Yjs document is represented by a `Y.Doc`. Changes to the document are encoded as binary `Uint8Array` updates, which may describe either incremental changes or the complete current state of a document. Yjs updates are designed to be commutative, associative, and idempotent: they may be applied in any order, combined repeatedly, or received more than once without changing the final converged result [19]. This makes encoded updates the basic unit through which Yjs replicas exchange and merge document state.

To determine what information one replica is missing from another, Yjs uses state vectors. A state vector compactly summarizes which inserted structures are already represented in a document state. For each client identifier, it records the next expected logical clock. For example, if a state vector records clock 5 for one client, then the document already contains that client's insertions with clocks 0 to 4, and the next insertion not yet known from that client would have clock 5 [19,24]. Given a state vector, `Y.encodeStateAsUpdate` can produce only the part of a live `Y.Doc` that is absent from another replica. Yjs also provides equivalent operations directly on already encoded updates: `Y.encodeStateVectorFromUpdate` derives a state vector from an encoded update, while `Y.diffUpdate` computes the portion of one encoded update that is missing with respect to a given state vector [19].

These primitives are also used during the standard Yjs synchronization handshake. When two peers first synchronize, one side can send a state vector indicating the document state it already has, and the other side responds with the missing update needed to bring it up to date [23]. After this initial reconciliation, providers such as `y-websocket` can propagate subsequent document changes as ordinary Yjs update messages [22].

2.2.3 Use of Yjs Primitives in the Prototype

The prototype in this thesis does not use the standard Yjs provider-based synchronization process directly. Instead, it uses the lower-level update primitives described above to synchronize through the DSM. The value stored under the document's DSM key is a full-state Yjs update

produced by `Y.encodeStateAsUpdate(ydoc)`. In the system design, this stored full-state update is referred to as a document snapshot because it captures the complete current document state under a single key.

When a client reads a newer snapshot from the DSM, it does not replace its local document. Instead, it first computes the state vector of its current local `Y.Doc` using `Y.encodeStateVector`. It then uses `Y.diffUpdate` to extract only the part of the remote snapshot that is still missing locally, and integrates that missing state with `Y.applyUpdate`. In this way, a client can incorporate newly persisted remote edits without reapplying state that it already knows.

The prototype derives a state vector from the latest snapshot stored in the DSM using `Y.encodeStateVectorFromUpdate` and uses it as a reference when checking whether the local document contains inserted structures that are not yet represented in that stored snapshot. When the client needs to construct a new value for the DSM, it combines the latest remote full-state update with its current local document state and encodes the merged result as a new full-state update. The backend therefore stores one repeatedly replaced encoded document state rather than a log of individual Yjs operations.

2.2.4 Why Yjs Fits the Proposed Architecture

These properties make Yjs particularly suitable for the architecture investigated in this thesis. Since encoded updates can be merged and applied by the clients, the backend does not need to understand the Yjs document model, execute the Yjs synchronization protocol, or inspect the binary payload it stores. A full-state Yjs update can be treated by the DSM as an opaque value, while each browser client remains responsible for decoding remote state, merging it with its local document, and writing back a new combined state when necessary.

The result is not that communication becomes unnecessary, but that Yjs-specific convergence logic can remain at the replicas. Repeated read, merge, and apply cycles remain safe because Yjs updates are idempotent and because replicas that eventually receive the same updates converge to the same state [15, 19]. This is the key reason Yjs is a suitable foundation for the proposed system: it allows the thesis to replace a dedicated CRDT-aware synchronization server with a generic replicated DSM backend while still preserving document-level convergence.

2.3 Quill Rich-Text Editor

Quill is a browser-based rich-text editor used in the prototype to provide the visible editing interface through which users create and observe document changes [13].

In the prototype developed for this thesis, Quill is used only as the user-facing editor component. The collaborative document state itself is held in a Yjs shared type, `Y.Text`, and the `y-quill`

binding connects that shared Yjs text object to the Quill editor instance. The binding maps `Y.Text` to Quill and keeps the two synchronized, so that local edits made through the editor update the Yjs document and remote changes applied to the Yjs document are reflected in the visible editor contents [18, 21].

Quill therefore does not participate in the synchronization architecture investigated in this thesis. It provides the editing interface through which users create and observe document changes, while Yjs is responsible for representing and merging collaborative state and the DSM is responsible for storing the encoded document value. This separation is useful for the prototype because it keeps the presentation layer independent from the synchronization mechanism being evaluated.

2.4 Distributed Shared Memory

2.4.1 Atomic Distributed Shared Memory

A DSM emulation provides clients with the abstraction of accessing shared read/write objects even though the data are physically maintained by a set of communicating servers in a distributed system [2]. In an atomic DSM, read and write operations must behave as if they were executed on a single shared memory object. This strongest consistency guarantee is commonly referred to as atomicity or linearizability [6, 7].

The purpose of an atomic DSM algorithm is therefore to hide the complexity of distributed replication from the clients. Although object values may be stored on several servers and messages may be delayed or reordered, clients should observe a shared object whose behaviour is indistinguishable from that of a single sequential memory location. At the same time, the system should tolerate failures of some servers and continue to complete read and write operations whenever the algorithm's fault assumptions are satisfied [2].

Early atomic DSM emulations assume a static setting, in which the set of servers storing the object remains fixed throughout the execution. Later reconfigurable algorithms allow this set to change over time, so that failed servers can be replaced or new servers can be added while preserving the shared-memory abstraction. This distinction is relevant to the present thesis because the backend used by the prototype comes from this line of work: ABD [2] is the foundational static atomic DSM emulation, ARES [10] extends atomic storage to a reconfigurable setting, and the coverable version [4, 11] used in the final system adds version-aware write semantics on top of that storage model.

2.4.2 The ABD algorithm

The ABD algorithm, introduced by Attiya, Bar-Noy, and Dolev, is the foundational fault-tolerant emulation of atomic shared memory in an asynchronous message-passing system [2]. In its

original form, it implements a single-writer, multiple-reader register in a static setting, where the set of replica servers is fixed and the algorithm remains correct provided that a majority of the servers do not crash [2]. Each stored value is associated with a logical timestamp, which allows replicas to distinguish newer values from older ones.

ABD relies on replication and majority quorums. A write operation completes in one communication round: the writer increases its local timestamp, sends the new timestamp–value pair to all servers, and terminates after receiving acknowledgments from a majority of them. A read operation requires two rounds. First, the reader queries the servers and waits for replies from a majority, selecting the value with the highest timestamp among the responses. Second, it propagates that highest timestamp–value pair back to a majority of servers before returning it to the client [2].

The second phase of the read operation is essential for preserving atomicity. Since any two majorities intersect, a read that follows a completed write is guaranteed to contact at least one server that stores the written value. By writing back the most recent value it has observed, the reader ensures that this value is itself stored by a majority before the read completes, so that later operations cannot return an older value. In this way, ABD makes a replicated collection of servers behave as a single atomic read/write object despite message delays and server crashes [2].

The original ABD algorithm assumes a single writer. Later extensions generalized the approach to multiple writers by replacing simple timestamps with tags that combine a logical timestamp and a writer identifier, so that concurrent writes can still be totally ordered [8]. For the purposes of this thesis, ABD is important because it provides the basic quorum-based atomic storage mechanism on which later systems build. In particular, ARES can use ABD-style data-access primitives as one of its underlying storage implementations, and the coverable backend used in the final prototype extends this lineage with version-aware write semantics.

2.4.3 ARES and Reconfigurable Atomic Storage

ARES extends atomic storage from the static setting of ABD to a reconfigurable environment, in which the set of servers responsible for storing the object may change over time while read and write operations continue [10]. This is useful in long-running distributed systems, where failed servers may need to be replaced, obsolete servers removed, or new servers added without interrupting access to the shared object. ARES is a reconfigurable atomic storage framework designed for this setting.

ARES organizes the system into a sequence of configurations, where each configuration identifies the servers currently responsible for storing the object and the mechanism used to access them. Its key design feature is modularity: rather than fixing one particular storage algorithm

internally, ARES defines a small set of Data Access Primitives (DAPs) through which configurations are accessed. These primitives include obtaining the highest known tag, obtaining the highest tag–value pair, and propagating a tag–value pair to the servers. Different storage mechanisms can be expressed through these primitives; in particular, ARES can be instantiated with ABD-style DAPs or with erasure-coded DAPs [10].

A reconfiguration operation introduces a new configuration and transfers the latest known object state to it before that configuration becomes finalized. Read and write operations must therefore account not only for the current object value but also for possible changes in the configuration sequence. At a high level, an operation first discovers the relevant configurations, obtains the highest tag or tag–value pair from them, and then propagates its result to the latest configuration before completing. This allows ARES to preserve atomicity even while servers are being added or removed and while different configurations may coexist during an execution [10].

For the purposes of this thesis, the most relevant aspect of ARES is not its erasure-coding capability but its role as the reconfigurable DSM framework on which the employed backend is based. The earlier version of the prototype used an ARES instance with ABD-style data-access primitives, while the final system uses its coverable counterpart. The next subsection therefore focuses on coverability, the additional version-aware write semantics that are essential to the synchronization algorithm developed in this thesis.

2.4.4 Coverability and Coverable ARES

Atomicity ensures that read and write operations appear to occur in a single sequential order, but it does not by itself prevent a writer from overwriting a value that has been updated since the writer last observed it. For objects whose new value is expected to be based on the current one, it is useful to associate each value with a version and require writers to update only the version they have actually seen. Coverability is a consistency condition for such versioned objects. A read returns both the current version and the associated value, while a write attempts to revise a specific version. If the version supplied by the writer is still the latest one, the write succeeds and creates a newer version; if the object has already advanced to a newer version, the write does not take effect and instead returns the latest version and value [11].

The difference can be illustrated with two concurrent writers. Suppose both clients read version v_1 of an object and then attempt to write new values based on it. If one write succeeds first, it produces a newer version v_2. When the second writer later tries to write based on the now-stale version v_1, a coverable object rejects that update rather than allowing it to overwrite the value associated with v_2. The second writer learns the newer version and value instead. Coverability therefore preserves the linearizable behaviour of the object while additionally preventing successful writes from extending obsolete versions [11].

The coverable extension of ARES, referred to as CoAres [4], applies this idea to the reconfigurable setting. CoAres keeps the reconfiguration protocol and the underlying data-access primitives of ARES, but modifies the write operation so that the client first obtains the latest tag–value pair and compares the discovered tag with its locally known version. Only when the two match does the writer generate a new, larger tag and propagate its new value; otherwise, the write is reported as unsuccessful and returns the latest pair without changing the stored object [4]. In this way, CoAres preserves version-aware writes even while the set of storage servers may change over time.

The final prototype in this thesis uses a coverable ARES instance with ABD-style data-access primitives as its DSM backend. This property is central to the design: when several clients attempt to replace the same stored document value concurrently, a client whose write is based on an outdated version cannot silently overwrite a newer value already accepted by the DSM. Instead, the backend exposes the conflict by rejecting the stale write and returning the latest stored value, leaving the client-side synchronization algorithm to merge the states and decide what to write next.

Chapter 3

Design and Implementation

3.1 Conventional Yjs Architecture and Design Objective 15
3.2 DSM Interface Available to the Prototype 16
3.3 Design Evolution 17
3.4 Final System Architecture and Document Storage Model 19
3.5 Snapshot-based Synchronization Algorithm 22

3.1 Conventional Yjs Architecture and Design Objective

Before presenting the designs explored in this thesis, it is useful to establish the conventional Yjs architecture from which they depart. Yjs provides convergence for replicated documents once updates have been exchanged and applied, but replicas still require a communication layer through which those updates can be propagated. In the conventional Yjs architecture considered here, synchronization is provided by the y-websocket provider. This provider follows a client-server model: clients connect to a shared WebSocket endpoint, and the server relays document updates and awareness information among the connected clients [22].

3.1.1 Conventional Yjs Client-server Architecture

Figure 3.1 shows the conventional topology at a high level. Each client maintains its own local Yjs document and connects to the same WebSocket server. Clients do not communicate directly with one another. Instead, all synchronization traffic passes through the server.

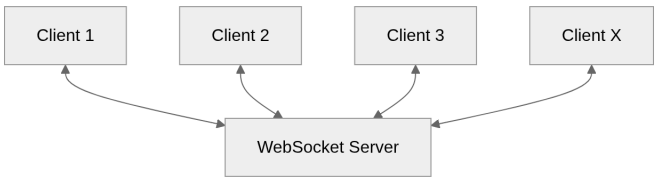


Figure 3.1: Conventional Yjs client-server architecture.

This arrangement provides a simple centralized communication layer through which a collaborative editing session can be maintained.

3.1.2 Conventional Update Propagation

Figure 3.2 shows the path of a local edit in this architecture. A change produced at one client is encoded as a Yjs update, sent to the WebSocket server by the `y-websocket` provider, and forwarded to the other connected clients participating in the same document session. Those clients then apply the received update to their own local `Y.Doc` instances [22, 23].

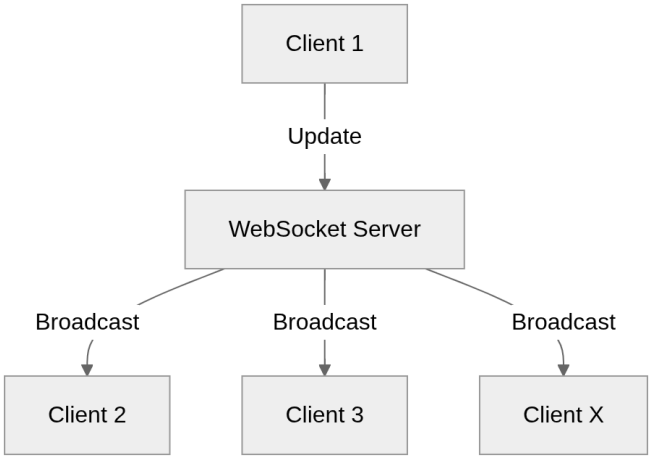


Figure 3.2: Propagation of a document update through a conventional Yjs WebSocket server.

This update-distribution role of the server is the aspect of the conventional architecture reconsidered in this thesis.

3.1.3 Design Objective of the Proposed System

The objective of the proposed system is to investigate whether collaborative editing can be supported without a dedicated Yjs-aware synchronization server. Instead of using a backend that receives and forwards Yjs update messages, the system aims to use a DSM as the only backend, while retaining Yjs at the clients for document representation and CRDT merge semantics.

The remainder of this chapter presents the successive designs explored to realize this objective and explains why the final implementation adopts a snapshot-based synchronization strategy over the coverable DSM.

3.2 DSM Interface Available to the Prototype

The DSM service used in this thesis is accessed through an HTTP API. The interface exposes key-value operations over stored objects, together with administrative operations for reconfig-

uration. The editor prototype uses only the data-access operations listed in Table 3.1. The reconfiguration-related operations exposed by the service are outside the scope of the editor implementation.

Table 3.1: DSM operations relevant to the prototype.

Operation	Purpose
`write`	Stores a value under a key. In the regular DSM variant the write succeeds unconditionally; in the coverable variant it may be rejected if the client is attempting to update a stale version.
`read`	Retrieves the value currently associated with a key.
`list`	Returns all stored keys, or only the keys matching a supplied regular expression.
`delete`	Removes a batch of specified keys from the store.

Each request includes a `clientID`, which identifies the client process issuing the operation. Values are transmitted as JSON-compatible payloads. Since Yjs updates are binary `Uint8Array` values, the prototype encodes them as base64 strings before writing them to the DSM and decodes them back into `Uint8Array` values after reading. This encoding step is independent of the storage design: in the initial design the encoded value represented an incremental Yjs update batch, while in the snapshot-based designs it represented a full-state Yjs update.

Two variants of the DSM were used during development. The earlier prototype designs used the regular, non-coverable version of the DSM, in which writes are not rejected due to stale client state. The final design uses the coverable version. In the coverable version, a write based on an outdated client view does not replace the current object value. Instead, the operation returns an `UNSUCCESS` status together with the latest stored value, allowing the client to update its local view before retrying.

The `list` operation is particularly relevant to the initial designs described next. Since those versions represented a document as a growing collection of DSM keys, clients had to repeatedly list all keys belonging to a document in order to discover updates written by other clients.

3.3 Design Evolution

3.3.1 Initial Per-update Approach

The first prototype followed an append-only design that represented each document as a collection of separately stored Yjs updates. Rather than keeping one current value for a document, every client wrote each locally generated batch of edits to a new DSM key. The key format was

```
yjs:<docId>:u:<clientID>-<seq>
```

where `docId` identified the shared document, `clientID` identified the writer, and `seq` was a local sequence number incremented by that client on each flush. In this design, the set of keys associated with a document grew monotonically over time.

Local edits were captured through a `ydoc.on("update")` listener. Updates generated by the user were accumulated in a pending buffer and periodically merged into one Yjs update using `Y.mergeUpdates`, encoded as base64, and written under a newly generated key. Since every flush used a fresh key, no client attempted to overwrite a value written by another client. The implementation therefore used the regular, non-coverable DSM variant, in which writes were not rejected because of stale object versions, so the prototype did not require coverability checks or retry logic.

To receive remote changes, each client periodically polled the DSM using the `list` operation with a regular-expression prefix matching the document's update keys. The client maintained a local `seenKeys` set and read only keys that had not previously been observed. Each newly discovered value was decoded from base64 and applied to the local `Y.Doc` using `Y.applyUpdate`. Remote updates were deliberately applied in arbitrary order, which was safe because Yjs updates are commutative and idempotent [19]. On startup, a newly joining client followed the same procedure and replayed every update key already stored for the document before reaching the current state.

Although this approach was simple and closely resembled an update log, it scaled poorly over time. Each active client created a new DSM key on every flush interval, so the number of keys increased with both session duration and the number of writers. Under continuous editing, this meant that the number of stored update keys increased steadily with both session duration and the number of active clients. As more clients edited the document, the DSM had to store and expose an increasingly large set of update keys.

A late-joining client also had to read and apply the complete historical update log individually before becoming synchronized. In addition, the browser-side `seenKeys` set grew for the lifetime of the tab, and no automatic garbage-collection mechanism existed to remove old update keys after all clients had processed them.

These limitations showed that, although the DSM could be used as an append-only transport for Yjs updates, the resulting cost of discovery, replay, and storage increased continuously with the history of the document. This motivated the next design, which introduced snapshots in an attempt to bound startup cost while still retaining separately stored incremental updates.

3.3.2 Hybrid Snapshot-plus-updates Scheme

The next design kept the append-only update log but added full-state Yjs snapshots in an attempt to reduce the cost of joining an existing document. A snapshot was produced with `Y.encodeState`

`AsUpdate(ydoc)`, encoded as base64, and stored separately from the individual update keys. Snapshot keys used the separate prefix `yjs:<docId>:s:`, distinguishing them from update keys under `yjs:<docId>:u:`. The update-writing path therefore remained essentially unchanged: clients still batched local Yjs updates and wrote each batch to a fresh DSM key, while snapshots provided an additional compact representation of the document state at a particular point in time.

The main benefit of this hybrid scheme was during startup. Instead of reconstructing the document only by replaying every historical update, a joining client could first load the snapshot containing the largest amount of Yjs state. It then examined the update keys and applied only those whose contents were not already represented in the snapshot. This reduced the amount of historical update data that had to be applied.

However, the approach did not remove the underlying dependence on the append-only key set. Incremental updates were still written as separate DSM objects, clients still used the `list` operation to discover them, and the number of stored keys still increased over time. The prototype included a `Create snapshot` button, so snapshot creation was manual rather than automatic. Moreover, creating a snapshot did not remove older update keys whose contents were already included in it.

A more complete version of this design could have added automatic periodic snapshots, together with garbage collection for DSM update keys and local `seenKeys` entries. Such a mechanism would need to determine which updates were safely represented by a snapshot before deleting them, while avoiding races with concurrent writers.

The hybrid design improved late-joining client behaviour, but it also required the system to manage both snapshot state and update-log state. These limitations motivated a more radical simplification: rather than treating snapshots as occasional checkpoints over an ever-growing log, the final design stores only one current full-state snapshot per document and repeatedly replaces that single value.

3.4 Final System Architecture and Document Storage Model

After the limitations of the append-only and hybrid designs, the final implementation adopts a snapshot-only architecture. The system still uses Yjs for document representation and merging, but the backend is no longer treated as an update log. Instead, the DSM stores one current encoded document state per shared document.

3.4.1 Final DSM-based Architecture

Figure 3.3 shows the final architecture of the prototype. Each browser client contains three main components: the Quill editor, the local `Y.Doc`, and a custom synchronization layer. Quill

provides the editing interface, Yjs maintains the collaborative document state, and the synchronization layer is responsible for reading from and writing to the DSM.

Unlike the conventional WebSocket architecture described earlier, clients do not send updates to a synchronization server that broadcasts them to other clients. Instead, each client communicates directly with one DSM node through HTTP `read` and `write` requests. The DSM backend is responsible for maintaining the replicated shared value internally, while the browser clients are responsible for all Yjs-specific processing.

The DSM therefore remains application-agnostic: it does not decode Yjs updates, maintain a `Y.Doc`, inspect the document structure, or perform CRDT merging. From the client's point of view, the coverable DSM acts as a linearizable replicated storage layer with version-aware writes, while all document-level merge logic remains inside Yjs.

3.4.2 Document Storage Model

In the final design, each collaborative document is stored under a single DSM key of the form

`yjs:<docId>:snap`

where `docId` identifies the shared document. The suffix `:snap` distinguishes the final snapshot-based representation from the earlier append-only update keys.

The value stored under this key is a full-state Yjs update:

`Y.encodeStateAsUpdate(ydoc)`

It is a binary Yjs update containing all state currently known to that local `Y.Doc`.

As described in Section 3.2, this binary update is encoded as base64 before storage and decoded back into a `Uint8Array` after reading.

The important property of this storage model is that the DSM contains only the latest full-state representation of the document. There is no append-only update log, no need to list document keys, and no historical update sequence that must be replayed by a joining client. A client only needs to read the current snapshot key for the document.

This single-key design makes storage and discovery simple, but it means that clients may concurrently try to replace the same stored value. For this reason, the final implementation uses the coverable DSM. A write based on an outdated version is rejected rather than silently overwriting a newer snapshot. The rejected client receives the latest stored value, merges it with its local Yjs state, and retries through the synchronization algorithm described in the next section.

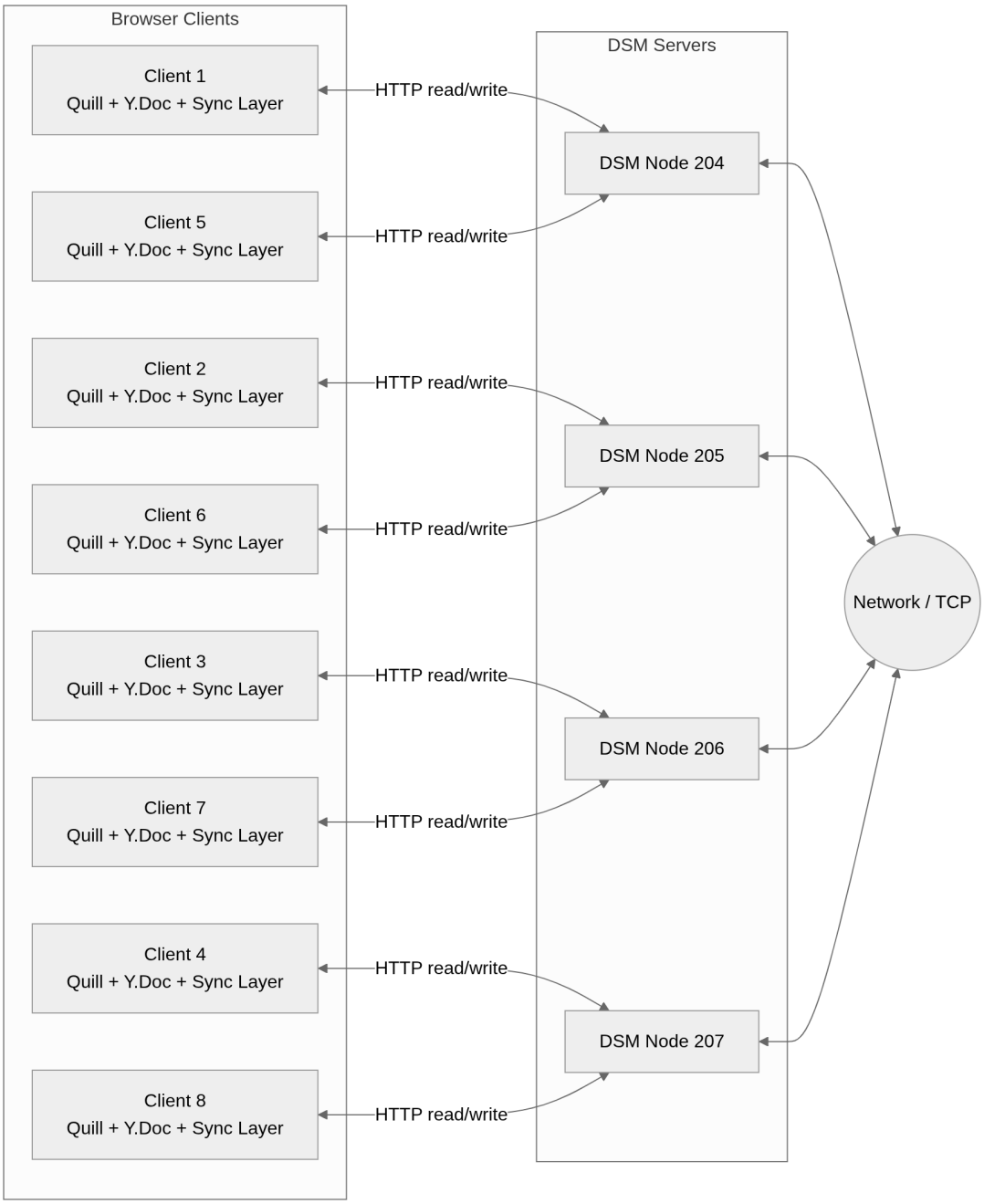


Figure 3.3: Final DSM-based collaborative editing architecture.

3.5 Snapshot-based Synchronization Algorithm

The final prototype synchronizes clients by repeatedly reading, merging, and conditionally writing full-state Yjs snapshots. Each client independently observes the latest stored snapshot, integrates any missing remote state into its local `Y.Doc`, and writes a new snapshot only when its local document contains changes that have not yet been persisted.

3.5.1 Algorithm Overview

When a client starts, it creates a local `Y.Doc`, binds it to the Quill editor, and reads the snapshot key associated with the selected document. If no snapshot exists, the client initializes the DSM entry by writing an empty Yjs state. Otherwise, it applies the stored snapshot locally and records its state vector in `knownDsmSV`, marking it as the latest DSM state known to the client.

The implementation maintains three main pieces of synchronization state:

- `latestRemoteSnapB64`, which stores the most recent snapshot read from the DSM
- `previousSnapB64`, which stores the most recent snapshot already processed by the local client
- `knownDsmSV`, which stores the state vector of the latest DSM snapshot known to the client, whether that snapshot was read from the DSM or written by the client itself

After initialization, the client starts two periodic loops. The first loop polls the DSM and stores the latest snapshot in `latestRemoteSnapB64`. The second loop is the sync loop. It first incorporates any remote state that is present in the latest DSM snapshot but missing from the local `Y.Doc`. It then checks whether the local document contains edits that are not yet represented in the DSM snapshot. If such edits exist, the loop creates a new full-state snapshot that combines the latest DSM snapshot with the client's current local `Y.Doc`, and attempts to write it back to the DSM.

Figure 3.4 summarizes the client initialization process before regular synchronization begins.

This separation is intentional. The polling loop only observes the DSM and does not modify the local Yjs document. All modifications to the local document and all write attempts are handled by the sync loop, which is protected by a `syncing` flag so that two sync iterations do not overlap.

3.5.2 Polling Loop: Reading the Latest DSM Snapshot

The polling loop is implemented by the `startPolling` function. At each interval, it calls `readSnapshotB64`, which reads the document's snapshot key from the DSM using the HTTP `/read` endpoint. If the read succeeds and returns a value, the loop stores the returned base64 snapshot in `latestRemoteSnapB64`.

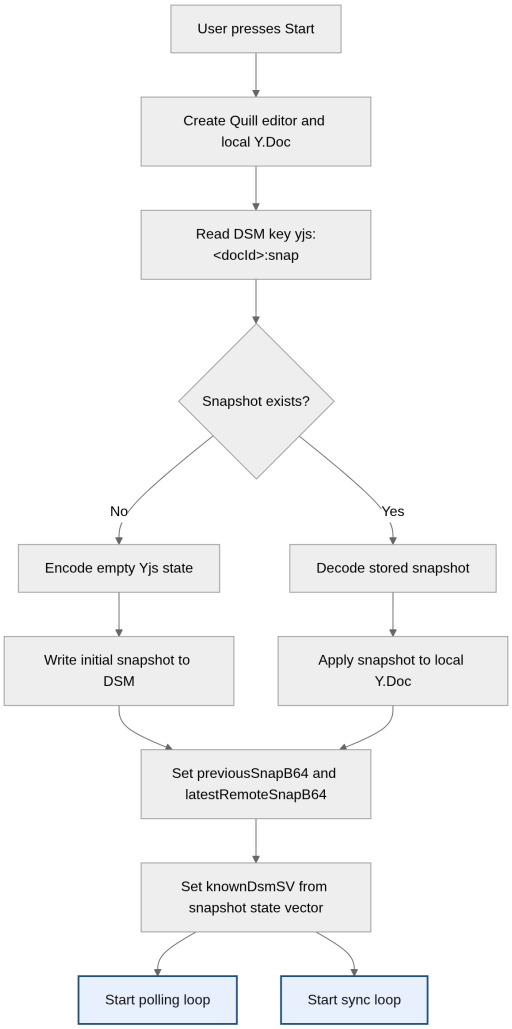


Figure 3.4: Client initialization and startup of the snapshot synchronization loops.

Figure 3.5 summarizes this process. Each iteration reads the single snapshot key for the document and, when a value is returned, records it as the latest remote snapshot observed by the client. The implementation also uses a local `polling` flag to prevent a new polling iteration from starting before the previous DSM read has completed.

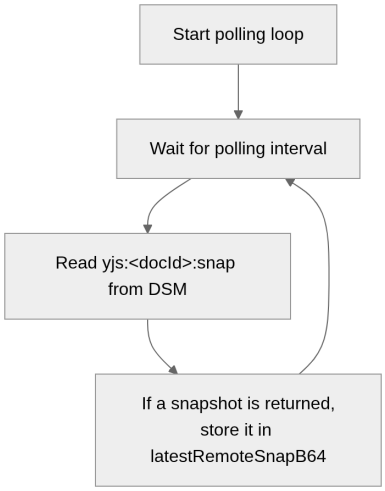


Figure 3.5: Polling loop for reading the current DSM snapshot.

The polling loop does not immediately apply the snapshot to the local `Y.Doc`. Its role is only to copy the snapshot returned by the DSM into `latestRemoteSnapB64`. This keeps all modifications to the local `Y.Doc` inside the sync loop.

This differs from the earlier append-only design, where polling required a `/list` request followed by several individual reads for unseen update keys. In the final design, polling always targets the single snapshot key of the document.

3.5.3 Sync Loop: Applying Remote State and Flushing Local Changes

The sync loop is implemented by the `startSyncLoop` function. Unlike the polling loop, which only reads from the DSM, the sync loop is responsible for modifying the local `Y.Doc` and for attempting writes back to the DSM. Each iteration performs two tasks in sequence: first, it applies any remote state observed by the polling loop that is missing locally; second, it attempts to flush local changes if the local document contains state that is not yet represented in the DSM snapshot.

The first part of the loop compares `latestRemoteSnapB64` with `previousSnapB64`. If the values differ, the client has observed a DSM snapshot that it has not yet processed. The snapshot is decoded from base64 into a `Uint8Array`, and the client computes its current local state vector using `Y.encodeStateVector(ydoc)`. It then uses `Y.diffUpdate` to extract only the part of the remote snapshot that is missing locally. If such a diff exists, it is applied to the local document

using `Y.applyUpdate`.

After processing the remote snapshot, the client updates `knownDsmSV` using `Y.encodeStateVectorFromUpdate`. This makes `knownDsmSV` correspond to the snapshot that has just been processed. The client also updates `previousSnapB64`, marking the snapshot as processed.

The second part of the sync loop is responsible for flushing local changes. If the client detects that its local `Y.Doc` contains state beyond `knownDsmSV`, it constructs a new full-state snapshot. To do this, the implementation creates a temporary empty `Y.Doc`, applies the latest remote snapshot to it, then applies the current local document state, and finally encodes the combined result using `Y.encodeStateAsUpdate`. This produces the snapshot that the client attempts to write to the DSM.

Figure 3.6 summarizes the two-phase structure of the sync loop.

In simplified form, the sync loop performs the following logic:

```
periodically:
    # Phase 1: incorporate remote DSM state
    if latestRemoteSnapB64 differs from previousSnapB64:
        remoteU8 = b64ToU8(latestRemoteSnapB64)
        localSV = Y.encodeStateVector(ydoc)

        # Extract only the remote state missing locally
        missing = Y.diffUpdate(remoteU8, localSV)

        if missing is not empty:
            Y.applyUpdate(ydoc, missing)

        knownDsmSV = stateVectorFromSnapshotU8(remoteU8)
        previousSnapB64 = latestRemoteSnapB64

    # Phase 2: publish local state if needed
    if local document has unwritten changes:
        fullStateU8 = combine latest remote snapshot with local Y.Doc state
        attempt to write fullStateU8 to DSM
```

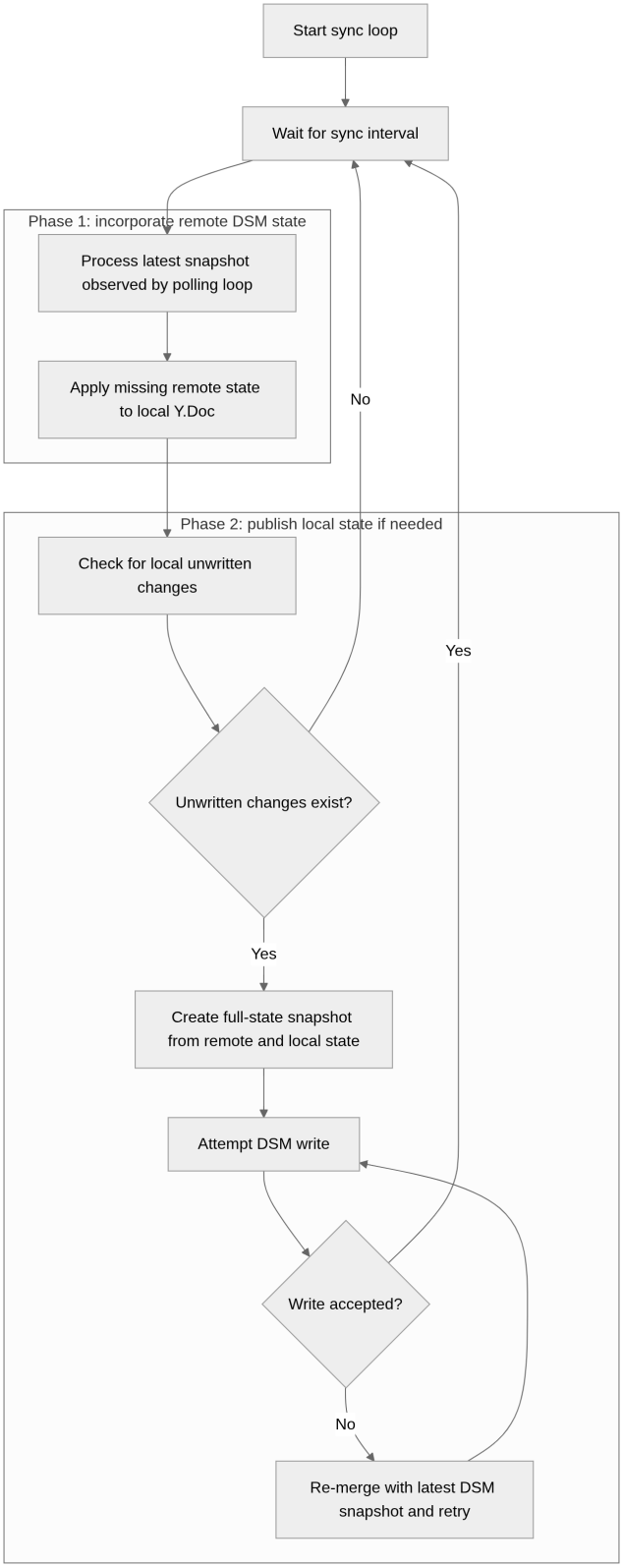


Figure 3.6: Two-phase sync loop for applying remote state and publishing local changes.

The write attempt may either succeed or be rejected by the coverable DSM if the client is writing based on an outdated view. The mechanisms for detecting local unwritten changes and handling rejected writes are described in the following subsections.

3.5.4 Detecting Local Unwritten Changes

The sync loop should not write a new snapshot to the DSM on every iteration. Most iterations only need to check whether anything has changed. A write is necessary only when the local `Y.Doc` contains document state that is not yet included in the latest DSM snapshot known to that client.

For this purpose, the implementation uses the variable `knownDsmSV`. This variable stores the state vector of the latest DSM snapshot that the client has already processed. It therefore acts as a boundary between two parts of the local document state:

- the state already represented in the DSM snapshot; and
- the state that exists locally but has not yet been written to the DSM.

In other words, `knownDsmSV` tells the client: "this is the DSM state that I know has already been persisted". If the local `Y.Doc` contains additional Yjs structures beyond that state vector, then the client has local changes that still need to be flushed to the DSM.

The implementation performs this check with the following function:

```
function localHasUnwrittenChanges() {
    if (!ydoc || !knownDsmSV) return false;

    const update = Y.encodeStateAsUpdate(ydoc, knownDsmSV);
    const { structs } = Y.decodeUpdate(update);

    return structs.length > 0;
}
```

The call

`Y.encodeStateAsUpdate(ydoc, knownDsmSV)`

asks Yjs to compute the part of the local document that is missing from the state described by `knownDsmSV`. In this system, that means: "compute the part of my local document that is not yet known to be stored in the DSM snapshot".

If the user has typed new text locally since the last known DSM snapshot, the resulting update contains Yjs internal structures. These structures appear in the decoded `structs` array. Therefore, when `structs.length > 0`, the client knows that it has local content that has not yet

been persisted, and the sync loop proceeds to construct and write a new full-state snapshot.

If `structs.length` is zero, then the local document does not contain any new inserted structures beyond the known DSM snapshot. In that case, the client skips the write phase for that iteration. This avoids repeatedly writing identical snapshots to the DSM when no local edit has occurred.

A simple example illustrates the purpose of the check. Suppose the latest DSM snapshot already contains all insertions up to a certain Yjs state vector. The client records this state vector in `knownDsmSV`. If the user then types a new word in the editor, the local `Y.Doc` advances beyond `knownDsmSV`. On the next sync iteration, `Y.encodeStateAsUpdate(ydoc, knownDsmSV)` returns an update containing that new insertion, so the function returns `true`. After the client successfully writes a merged snapshot to the DSM, `knownDsmSV` is updated to match the newly written snapshot, and the same insertion is no longer considered unwritten.

One limitation of this condition is that it detects only new inserted content. In Yjs, inserted content is stored as internal items, and these items are the structures that appear in the decoded `structs` array. Deletions are handled differently: deleting text does not create a new inserted item. Instead, Yjs marks existing items as deleted and includes the deleted item identifiers in the update's delete information [24]. Therefore, an update that contains only a deletion may have an empty `structs` array. Since the current function returns `true` only when `structs.length > 0`, a deletion-only edit may not trigger a DSM write.

This does not mean that the deletion is lost from the local `Y.Doc`. The local document state still contains the deletion information. However, because the write condition is not triggered, the deletion may not be propagated to the DSM immediately. If the same client later performs an insertion, that insertion creates new Yjs structures, causing `localHasUnwrittenChanges()` to return `true`. The sync loop then constructs a new full-state snapshot from the complete current local `Y.Doc`. Since this snapshot is generated from the entire local document state, it includes both the earlier deletion information and the later insertion. When the snapshot is written to the DSM, other clients will eventually receive and apply both changes. Thus, in the current implementation, a deletion-only edit may be delayed, but it can still propagate later as part of a subsequent snapshot triggered by an insertion.

3.5.5 Conflict Handling and Coverability Retries

After the client constructs a merged full-state snapshot, it attempts to write that snapshot to the DSM using the document's single snapshot key. Since the final design repeatedly replaces one stored value, concurrent clients may attempt to write different snapshots based on different previously observed versions of the document. This is where the coverable DSM is used.

If the write returns `SUCCESS`, the snapshot has been accepted by the DSM. The client then up-

dates its local synchronization state: `previousSnapB64` and `latestRemoteSnapB64` are set to the written value, `knownDsmSV` is updated to the state vector of the accepted snapshot. The implementation also applies any missing part of the accepted merged snapshot back to the local `Y.Doc`, in case the merged value included remote state that was not yet present locally.

If the write returns `UNSUCCESS`, the DSM has rejected the write because the client was not writing on top of the latest stored version. In that case, the response includes the latest stored snapshot. The client decodes this returned snapshot, merges it again with its current local `Y.Doc`, produces a new full-state snapshot, and retries the write.

This retry process is bounded by `MAX_RETRIES` in the implementation. Bounding the retry count also limits how long a client may remain inside a single sync-loop iteration. During the retry cycle, the implementation rebuilds merged snapshots using the newer DSM state returned by unsuccessful writes, but it does not immediately apply that returned state to the live local `Y.Doc`. Since the `syncing` flag prevents overlapping sync-loop executions, remaining indefinitely inside the retry loop could delay the incorporation of newer remote updates into the client's visible local document state.

In simplified form, the retry cycle is:

```
while attempts remain:
    result = write merged snapshot to DSM

    if result is SUCCESS:
        update local snapshot state
        stop

    if result is UNSUCCESS:
        remote snapshot = result.value
        merge remote snapshot with local Y.Doc
        retry with the new merged snapshot
```

The important point is that a rejected write is not treated as a permanent failure. Instead, it means that another client has already written a newer snapshot. The rejected client uses the returned snapshot as the new base, merges it with its current local `Y.Doc`, and retries.

Chapter 4

Demonstration of the Collaborative Editing Prototype

4.1 Prototype Interface Overview 30

4.2 Starting a Collaborative Session 31

4.3 Runtime Diagnostics and Synchronization Observability 34

4.4 Collaborative Editing Across Multiple Clients 35

This chapter presents a demonstration-oriented walkthrough of the collaborative editing prototype developed in this thesis. The purpose of this chapter is to provide a practical view of the system from the users' perspective and to illustrate how collaborative editing and synchronization are performed during runtime.

The prototype combines the Quill rich-text editor with a Yjs CRDT synchronization layer operating over the coverable DSM backend presented in the previous chapters. Multiple clients can join the same collaborative session by connecting to the same document identifier.

In addition to the collaborative editing functionality itself, the prototype includes a lightweight diagnostics panel that exposes internal synchronization activity in real time. The diagnostics panel was designed specifically for demonstration and evaluation purposes, allowing the runtime behavior of the DSM-based synchronization layer to become observable during execution. Through this interface, users can observe DSM read and write activity, synchronization retries, logical version progression, and document convergence while the system is running.

The following sections present the user interface of the prototype, the process of starting collaborative sessions, and a series of execution scenarios demonstrating synchronization across multiple clients.

4.1 Prototype Interface Overview

Figure 4.1 presents the main interface of the collaborative editing prototype. The interface is divided into three main parts: the session configuration panel, the collaborative rich-text editor, and the runtime diagnostics panel.

The left-hand panel allows the user to configure a collaborative editing session before it is started. It includes the DSM node endpoint, the shared document identifier, and the client identifier. The right-hand side contains the Quill-based rich-text editor, which is synchronized across clients through the DSM-based synchronization algorithm described in Section 3.5.

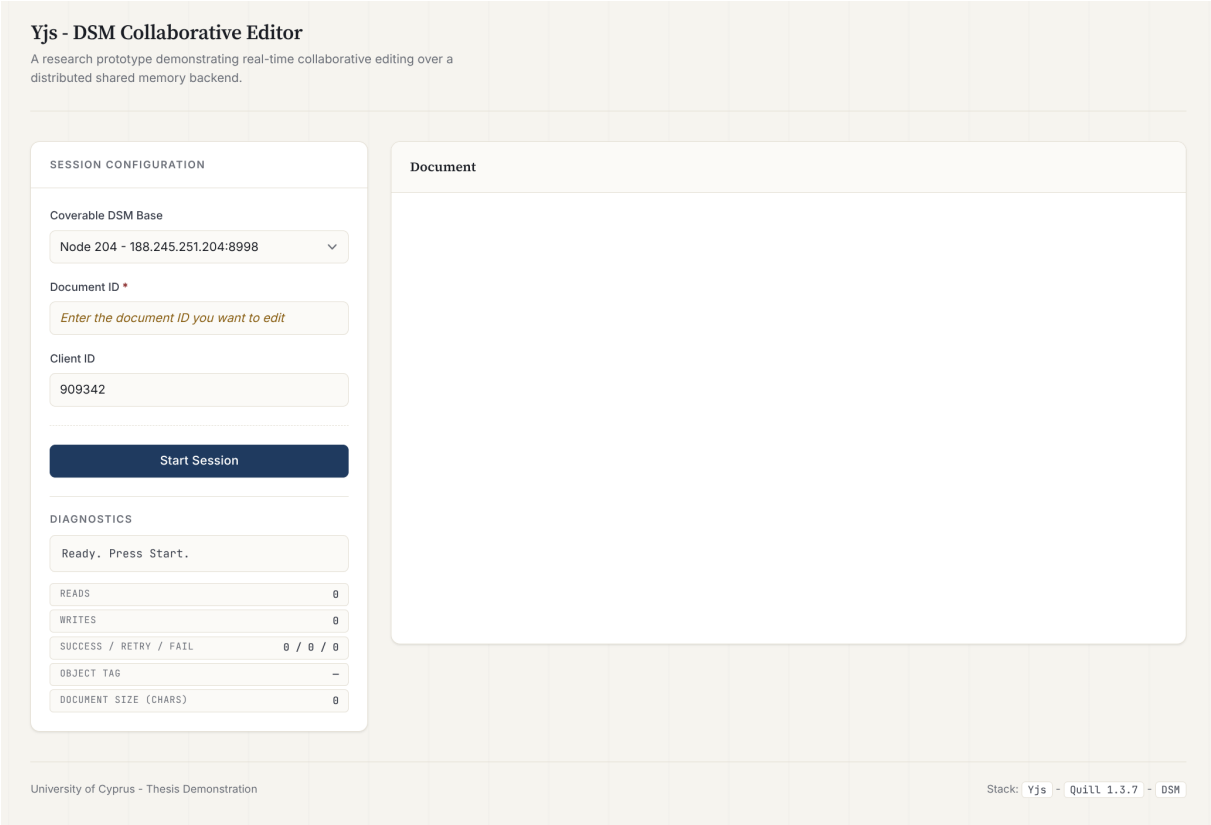


Figure 4.1: Main interface of the DSM-based collaborative editing prototype

The lower part of the configuration panel includes a lightweight diagnostics area. This panel was intentionally added to the demonstration version of the prototype in order to expose internal synchronization behavior during execution. It allows the user to observe DSM reads and writes, retry behavior, the DSM object tag, and the current visible document state in real time. The configuration controls and diagnostics values are discussed in more detail in the following sections.

4.2 Starting a Collaborative Session

This section presents the process of initializing a collaborative editing session using the prototype interface. In order to participate in a shared editing session, users must configure the synchronization endpoint and specify the document they wish to collaboratively edit.

Figure 4.2 illustrates the session configuration process before the collaborative session is started.

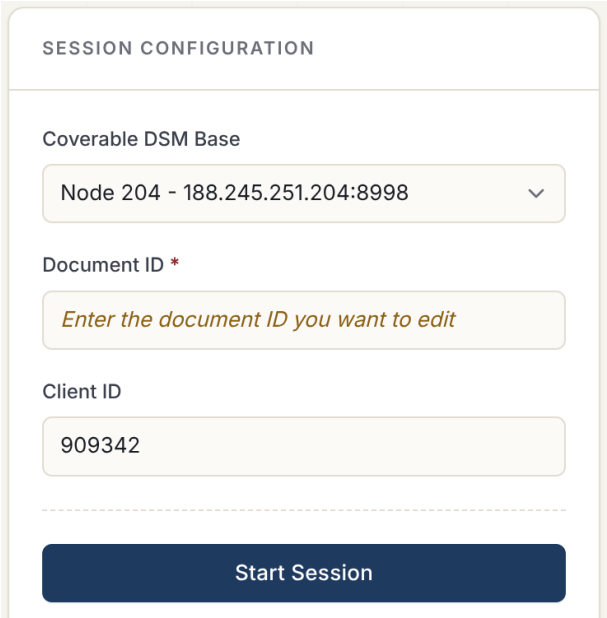


Figure 4.2: Collaborative session configuration before initialization

The DSM node endpoint is selected through the `Coverable DSM Base` dropdown menu as shown in Figure 4.3. Each option corresponds to a different DSM node. The selected node acts as the synchronization endpoint through which the client performs DSM read and write operations during collaborative editing.

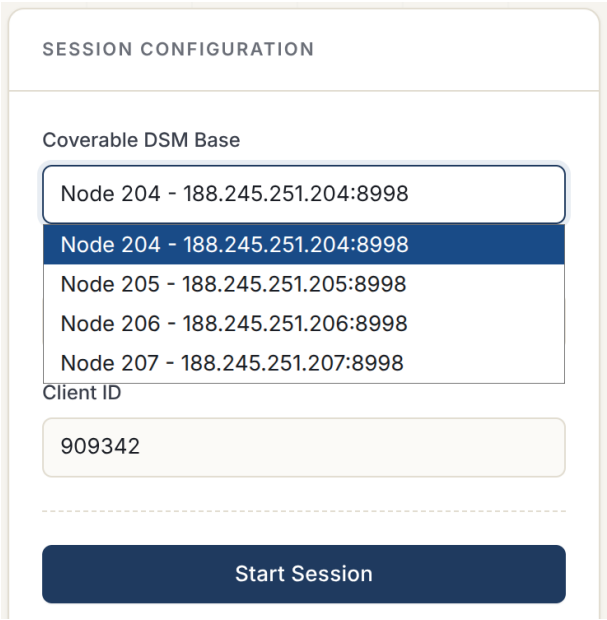


Figure 4.3: Coverable DSM Node selection dropdown menu

The user must also specify a document identifier. Clients that connect using the same document identifier automatically participate in the same collaborative editing session and synchronize

over the same shared Yjs document state. This mechanism allows multiple users to collaboratively edit the same document concurrently.

The client identifier represents the writer identity of the current browser session. When the page is loaded, the prototype automatically creates a random client ID and stores it in the browser, while still allowing the value to be manually modified before the session starts. This identifier is required for session initialization. It is important for the coverable DSM, since DSM tags include the writer identifier and write operations may be rejected if the client has not first observed the latest value of the object.

After the required configuration values are provided, the collaborative session may be initialized through the `Start Session` button. Once initialized, the client connects to the DSM backend, initializes the Yjs synchronization layer, and begins polling the DSM for remote updates.

Figure 4.4 presents the prototype after a collaborative session has been successfully initialized.

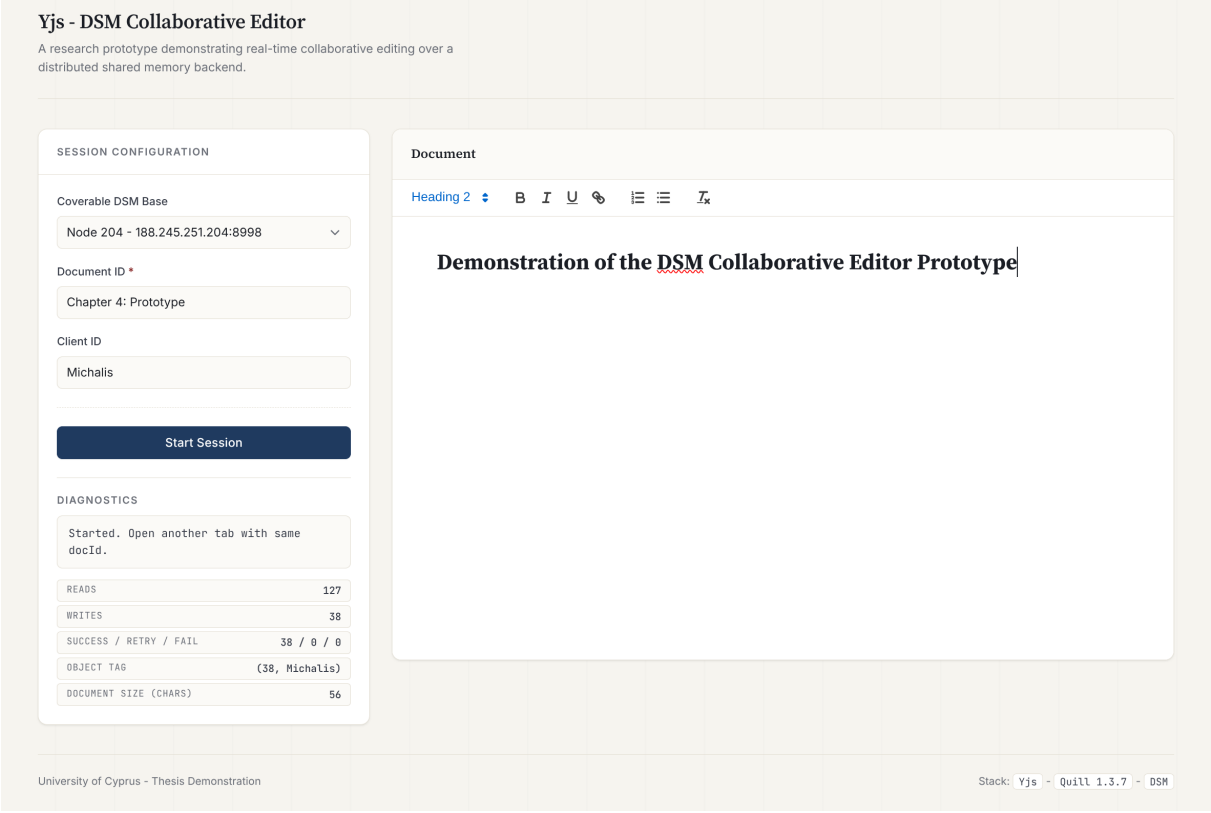


Figure 4.4: Collaborative editing session after successful initialization

After initialization, document state changes made by one client can become visible to the remaining connected clients through the DSM-based synchronization algorithm. At the same time, the diagnostics panel begins displaying runtime synchronization activity, including DSM reads, writes, retries, and object tag progression.

4.3 Runtime Diagnostics and Synchronization Observability

To make the execution of the synchronization algorithm easier to observe during the demonstration, the prototype includes a runtime diagnostics panel. This panel exposes selected internal values of the DSM-based synchronization layer while the collaborative editor is running.

For the demonstration version of the prototype, the polling and synchronization loops were configured with a 200 ms interval. This value was selected empirically during development, as it provided the most stable observed behaviour among the tested interval settings. It offered a practical balance between frequent synchronization attempts and avoiding excessive contention caused by overly aggressive polling and writing.

Figure 4.5: Runtime diagnostics panel of the collaborative editing prototype

As shown in Figure 4.5, the diagnostics panel reports the number of DSM read and write operations performed by the client, as well as the outcome of write attempts. Successful writes indicate that the DSM accepted the new value, while retries indicate that the client attempted to write based on an outdated view of the object and therefore had to first observe a newer value.

The panel also displays the current DSM tag observed by the client. In this implementation, the tag is represented as a pair (ts, wid), where ts is the logical timestamp and wid is the writer identifier. This makes the progression of DSM object versions visible during the demo. Finally, the document size provides a simple indication of the current visible state of the editor. When different clients report the same document size, this gives a useful first indication that their visible states are consistent, although it is not by itself a formal guarantee that the document contents are identical.

Overall, the diagnostics panel acts as a lightweight observability layer for the demonstration version of the prototype.

4.4 Collaborative Editing Across Multiple Clients

This section demonstrates the main collaborative editing behavior of the prototype using multiple clients connected to the same document. The scenarios presented below show how clients join the same collaborative session, how document state becomes visible across clients, and how the system behaves when concurrent modifications cause DSM write retries.

4.4.1 Multi-client Setup

In the demonstration scenario, two browser clients are started as separate user sessions. For example, one client may be served from `localhost:5173` and connected to DSM node 204, while another client may be served from `localhost:5174` and connected to DSM node 205. Although the clients may access the system through different local ports and different DSM nodes, they join the same collaborative editing session when they use the same document identifier.

Figure 4.6 shows two clients connected to the same shared document. Each client has its own client identifier, while both clients use the same document ID. This setup allows the users to edit the same document through separate browser sessions.

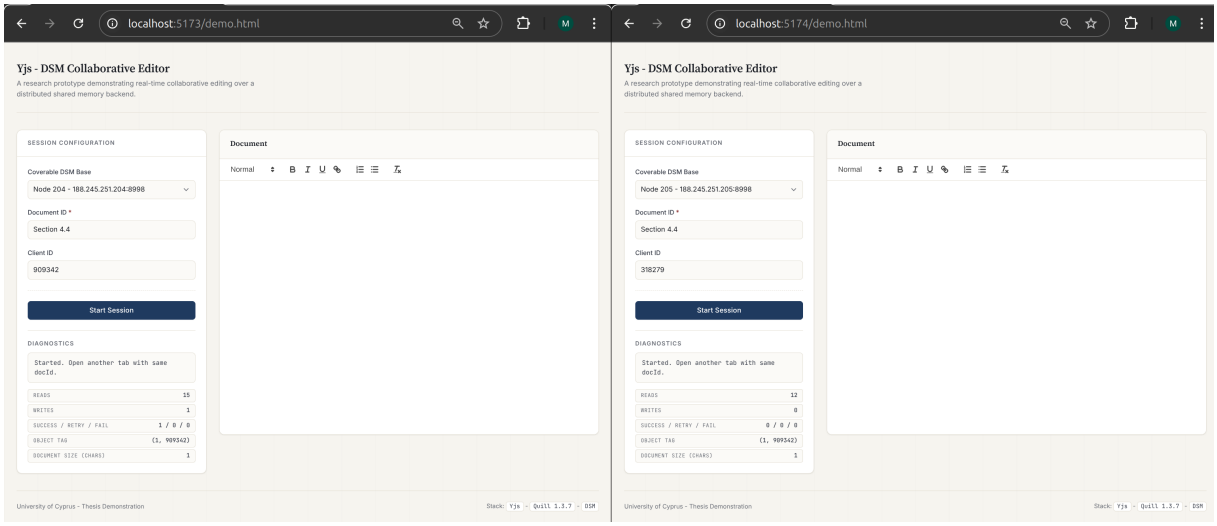


Figure 4.6: Two clients initialized on the same document identifier through different DSM nodes

4.4.2 Initialization of the Shared DSM Object

Figure 4.6 also illustrates the initialization behavior of the shared document. In this example, the left client is the first participant to access the document identifier. Since no corresponding DSM object exists yet for this document, the client encodes its initial empty Yjs state and writes it to the DSM. This is why the diagnostics panel of the left client shows one write operation, and the observed object tag becomes $(1, 909342)$.

The right client then joins the same collaborative session using the same document identifier. In this case, the DSM object already exists, so the client reads the stored value from the DSM and applies it to its local Yjs document state. As a result, both clients start from the same shared document state, even though they are served from different local ports and are connected to different DSM nodes.

4.4.3 Writing and Observing Shared Document State

Once both clients have joined the session, a modification performed by one client eventually becomes visible to the other through the shared DSM snapshot. Figure 4.7 shows an example where Client A inserts text into the editor. The local Quill editor and the corresponding Yjs document are updated immediately on Client A. The synchronization layer then detects that the local Yjs document contains state that has not yet been written to the DSM, encodes the current document state, and attempts to write the resulting snapshot to the DSM object.

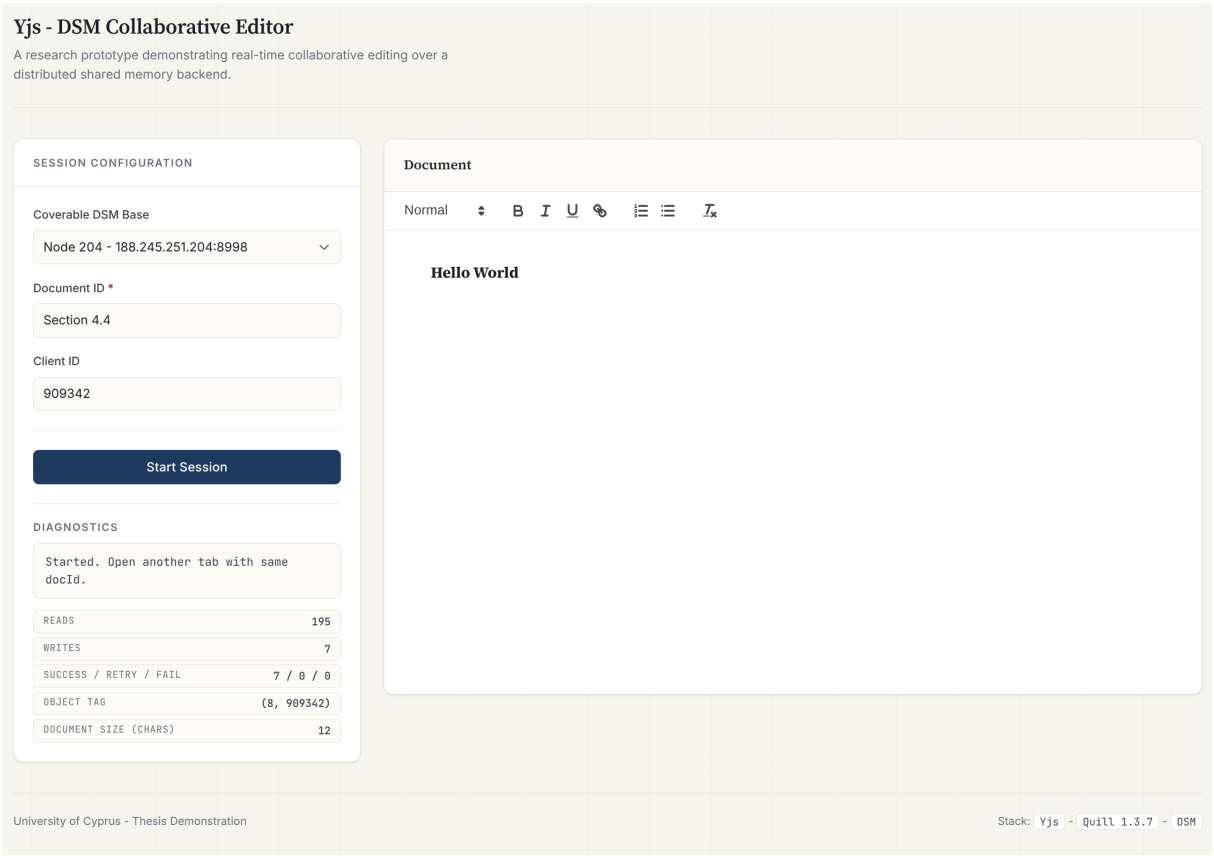


Figure 4.7: Client A inserting text into the shared editor

Figure 4.8 shows the corresponding state becoming visible on Client B. Client B periodically reads the DSM object, observes that a newer encoded document snapshot is available, computes the part of that snapshot that is missing from its local Yjs document, and applies it locally. As

a result, the visible editor content of Client B is updated to reflect the latest shared document state.

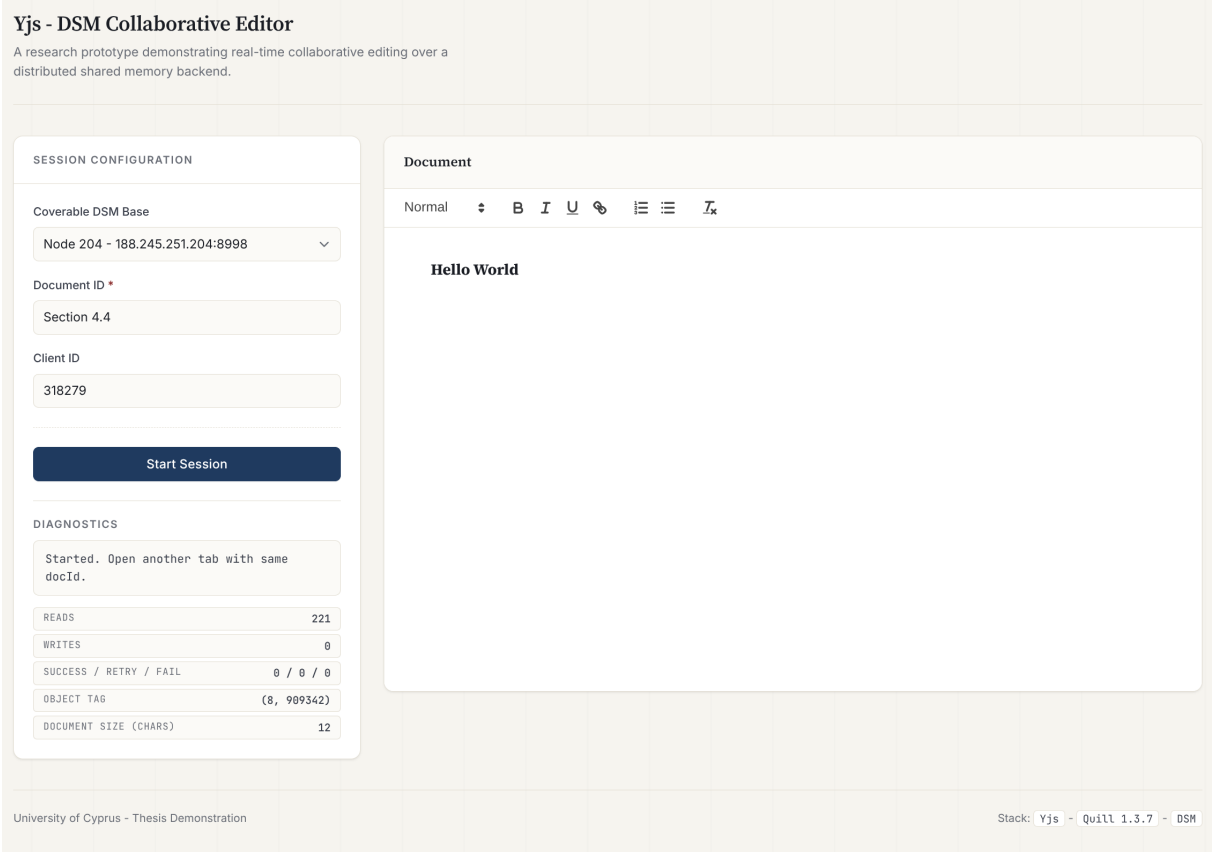


Figure 4.8: Client B observing the newer shared document state

4.4.4 Concurrent Editing and Retry Behavior

A more interesting scenario occurs when both clients modify the document concurrently. In this case, both clients may attempt to write newly encoded document snapshots based on the DSM value they have most recently observed. If one client writes successfully first, the other client may then attempt to write using an outdated view of the object. The coverable DSM rejects such stale writes, causing the synchronization layer to read the newer value, merge it with the local Yjs state, encode the merged state, and retry the write.

Figure 4.9 illustrates this behavior during concurrent editing. In this scenario, the two clients perform rapid text insertions in a short period of time. This increases the likelihood that one client attempts to write based on a DSM snapshot that has already been superseded by a successful write from the other client.

The diagnostics panel makes this process visible. The write counters increase, retry events appear, the DSM object tag advances, and the writer component of the tag changes according

to the client that produced the latest accepted snapshot. This provides direct visual evidence of concurrency, synchronization, and coverability-based conflict handling during runtime.

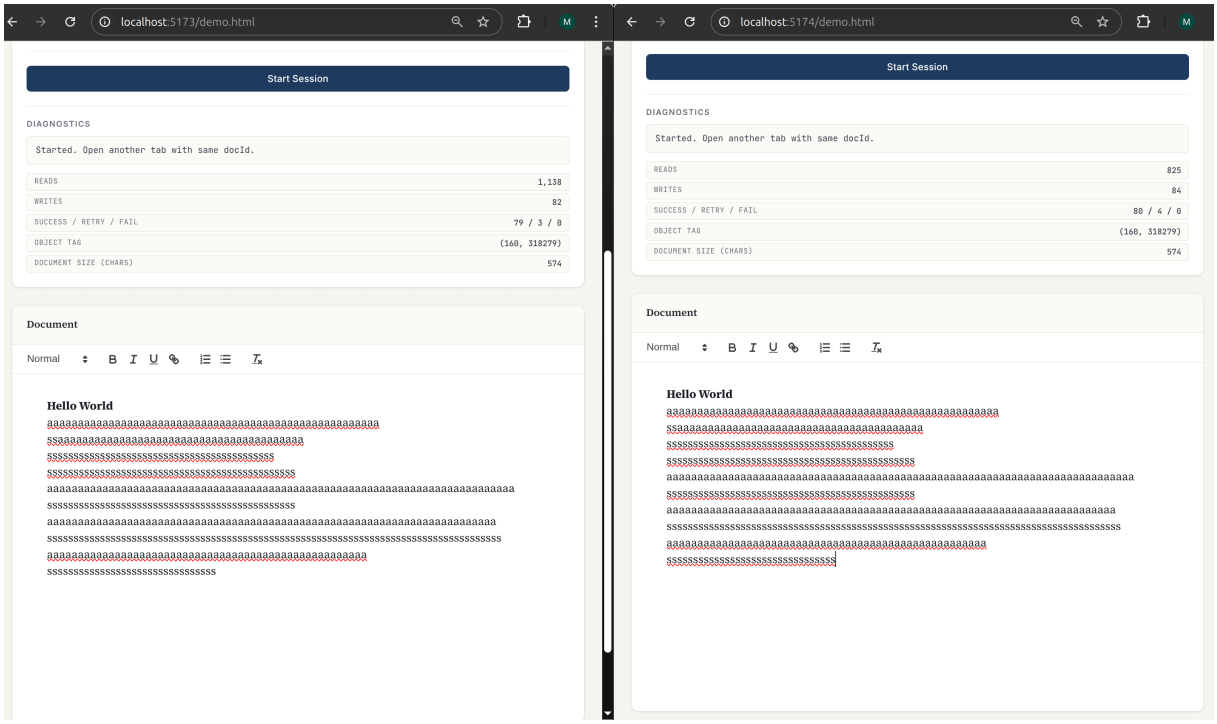


Figure 4.9: Concurrent text insertion from two clients, showing successful writes, retries, and DSM tag progression

In Figure 4.9, both clients eventually observe the same DSM object tag, indicating that they have read the same latest accepted DSM snapshot. The document size is also the same on both clients, which provides a useful sanity check that the visible editor states have likely converged, although this value alone does not formally guarantee that the document contents are identical.

4.4.5 Joining or Rejoining an Existing Session

Another useful demonstration scenario is to start a third client, or refresh one of the existing clients, after edits have already been made. When the client joins using the same document identifier, it reads the current DSM value and reconstructs the corresponding Yjs document state locally. This shows that the DSM object acts as the shared synchronization point for the collaborative document, allowing clients to join or rejoin an existing session and obtain the latest available state.

Overall, these scenarios demonstrate the complete user-facing behavior of the prototype. Users can connect through different browser sessions, select possibly different DSM nodes, join the same collaborative document using a common document identifier, and observe shared document state changes across clients.

Chapter 5

Experimental Evaluation

5.1 Evaluation Objectives 39
5.2 Experimental Setup 39
5.3 Benchmark Workload 40
5.4 Measured Metrics 40
5.5 Correctness Validation 41
5.6 Results and Discussion 41
5.7 Comparison with the Conventional `y-websocket` Architecture 47
5.8 Limitations 50

5.1 Evaluation Objectives

The purpose of the experimental evaluation is to assess both the correctness and the performance of the proposed DSM-based collaborative editor. Correctness is evaluated by checking whether all clients eventually converge to the same final document and whether all generated edits are preserved exactly once. Performance is evaluated by measuring read and write throughput, read and write latency, and write retry behaviour as the number of concurrent clients increases.

A second experiment then compares the proposed DSM-based architecture with a conventional `y-websocket` system.

5.2 Experimental Setup

The evaluation was performed using a Selenium-based [14] benchmark that launches multiple browser instances, each running the prototype editor. All clients join the same document by using the same document identifier, but each client is assigned a distinct client identifier.

Four local instances of the prototype web application were used, running on ports `5173`–`5176`. Browser clients were assigned to these application instances and to the four DSM nodes in round-robin fashion. Figure 3.3 illustrates this topology for the eight-client case.

The experiments were conducted with 1, 2, 4, 8, and 16 concurrent clients. Each configuration was repeated three times, and the reported results use the mean of the three runs. Standard deviation is shown on the relevant charts. Each run was divided into three stages. First, the clients were allowed to start and synchronize during a short warm-up period. Second, the bots inserted tokens for 30 seconds while performance metrics were recorded. Third, after the bots stopped, the benchmark waited until all clients converged to the same final document state. For the experimental evaluation, the client used an aggressive benchmark configuration, with both the polling loop and the synchronization loop scheduled every 1 ms.

Table 5.1: DSM stress-test benchmark configuration.

Parameter	Value
Concurrent clients	1, 2, 4, 8, 16
Runs per configuration	3
Application instances	4 local instances, ports `5173`–`5176`
Polling interval	1 ms
Synchronization interval	1 ms
Maximum write retries	40 attempts
Warm-up period	3 seconds
Measurement period	30 seconds
Bot insertion interval	10 ms per client
Client assignment	Round-robin across application instances and DSM nodes

5.3 Benchmark Workload

The workload is generated by an automated bot running inside each browser client. During the active measurement period, each bot repeatedly appends a token to the shared editor. Tokens have the form `Ck-nnnnnn`, where `Ck` identifies the client and `nnnnnn` is a monotonically increasing sequence number generated by that client.

This token format serves two purposes. First, it creates a continuous concurrent editing workload against the shared document. Second, it allows the benchmark to verify correctness after the run, because every token that was generated by a bot is known and can be searched for in the final document.

5.4 Measured Metrics

During each run, the benchmark collected both performance metrics and write outcome statistics. Read and write counts record the number of DSM read operations and DSM write attempts issued during the measurement period. Read and write throughput are reported as operations per second.

The benchmark also records read and write latency. For both operation types, latency is summarized using the average, median, p95, and p99 values. The percentile metrics are included because tail latency is important in collaborative editing: even if most operations complete quickly, occasional slow operations can delay the visibility of remote edits.

For write operations, the benchmark additionally distinguishes between successful writes, unsuccessful writes, and failures. A successful write is a write accepted by the coverable DSM. An unsuccessful write is a rejected write attempt caused by a stale snapshot version; such writes are retried by the client. A write failure is recorded only if the client exhausts the configured retry bound of 40 attempts without successfully writing the snapshot.

5.5 Correctness Validation

After the active measurement period ends, the benchmark stops the token-generating bots and waits for the clients to converge. Convergence is defined as all browser clients exposing exactly the same final editor text. Once convergence is detected, the benchmark extracts all tokens from the final text and compares them with the set of tokens actually generated by the bots.

A run is considered correct only if all clients have converged and every generated token appears exactly once in the final document, with no missing, extra, or duplicate tokens.

5.6 Results and Discussion

5.6.1 Correctness Results

All test configurations passed the correctness checks. Across the five DSM stress-test configurations and three runs per configuration, this corresponds to 15 successful runs out of 15. This demonstrates that, for the evaluated workload, the snapshot-based synchronization algorithm preserved all generated edits despite concurrent writes and coverability retries.

5.6.2 Aggregate Throughput vs Concurrent Clients

Figure 5.1 depicts the aggregate read and write throughput as the number of concurrent clients increases. Throughput increases with the number of clients because more clients are simultaneously polling the DSM and attempting to publish local edits. Read throughput increases from approximately 4.19 operations per second with one client to approximately 32.5 operations per second with sixteen clients, corresponding to a 7.8x increase. Write throughput follows a similar trend, increasing from approximately 4.25 to 33.5 operations per second, also corresponding to approximately a 7.9x increase.

However, this growth is sublinear. While the number of clients increases by 16x, aggregate

read and write throughput increase by less than 8x. Compared with ideal linear scaling from the one-client case, the sixteen-client configuration reaches only about 48.5% of the expected read throughput and about 49.3% of the expected write throughput.

This reflects both contention in the synchronization design, where multiple clients compete to update the same DSM snapshot key, and contention in the benchmark environment, where all browser instances execute on the same host.

Figure 5.1: Aggregate read and write throughput as the number of concurrent clients increases.

5.6.3 Write Outcome Composition vs Concurrent Clients

Figure 5.2 depicts the composition of write outcomes as the number of concurrent clients increases. With one client, all write attempts are accepted because there is no competing writer. As more clients write concurrently, the proportion of successful writes decreases, while the proportion of unsuccessful writes that must be retried increases. From four clients onward, retries become the dominant write outcome, reaching approximately 85% of write attempts at sixteen clients.

This behaviour is expected in the final design because all clients attempt to replace the same DSM snapshot key. As the number of concurrent writers increases, more clients attempt to write based on a snapshot version that has already been superseded by another client, causing the coverable DSM to reject the stale write and return the latest value instead.

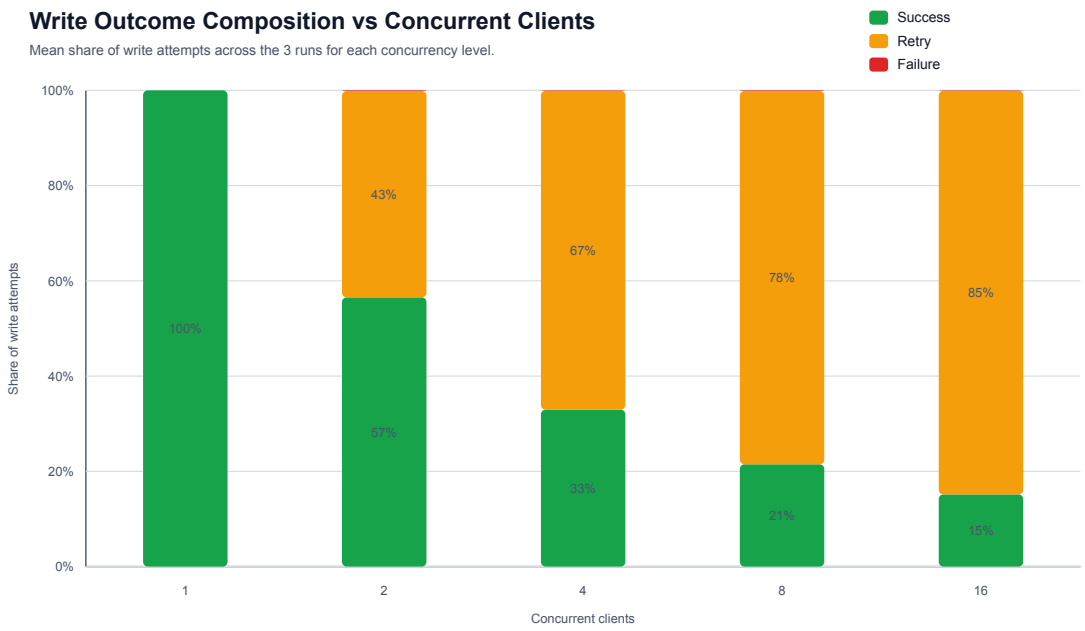


Figure 5.2: Write outcome composition as the number of concurrent clients increases.

5.6.4 Read Latency During the Active Writing Phase

Figure 5.3 depicts the read latency during the 30-second active writing phase. Each point represents the mean read latency for one measurement second, averaged across the three runs for the corresponding client count.

Two trends are visible. First, read latency generally increases with the number of concurrent clients. The one-client case remains the lowest throughout the run, staying roughly in the 200–300 ms range for most of the measurement period. The two-client and four-client cases are slightly higher, while the eight-client and sixteen-client cases show the highest read latencies. The higher-concurrency configurations frequently exceed 500 ms, and the sixteen-client case reaches peaks of approximately 800 ms.

Second, read latency tends to increase over time during the active writing period, especially for the higher-concurrency configurations. This suggests that the benchmark becomes more loaded as clients continue polling, editing, merging, and writing snapshots. The effect is also reflected in the larger error bars for the eight-client and sixteen-client cases, which indicate greater variability across the three runs. However, this figure measures end-to-end latency from the browser clients, so the observed increase cannot be attributed only to the DSM backend. It also includes client-side effects such as browser scheduling, Selenium overhead, and contention from running several active browser instances on the same machine.

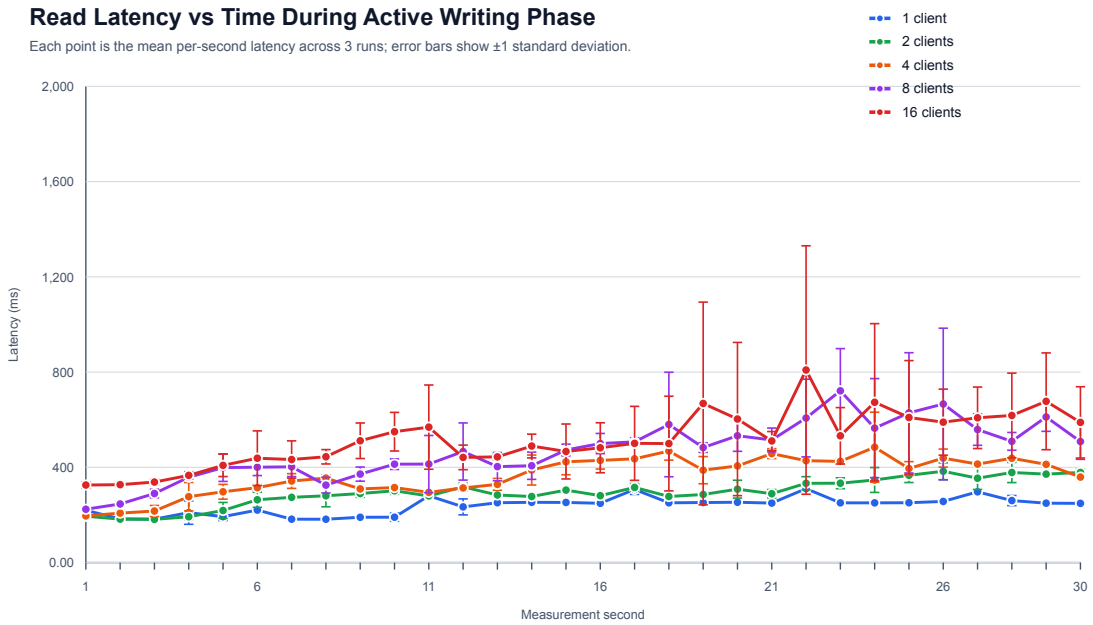


Figure 5.3: Read latency during the active writing phase.

5.6.5 Write Latency During the Active Writing Phase

Figure 5.4 depicts the write latency during the 30-second active writing phase. Each point represents the mean latency of DSM write attempts during one measurement second, averaged across the three runs for the corresponding client count. This metric includes both writes that returned `SUCCESS` and writes that returned `UNSUCCESS`; it therefore measures the latency of individual write attempts, not the total time required for a client to eventually complete a successful write after retries.

The chart shows that write latency generally increases with the number of concurrent clients. The one-client and two-client configurations remain comparatively low, with most values staying below approximately 400 ms. The four-client case is higher and the eight-client and sixteen-client cases show the largest latencies. In particular, the sixteen-client configuration rises from roughly 320 ms at the beginning of the measurement period to around 600 ms near the end.

A second trend is visible over time: write latency tends to rise during the active writing period, especially in the higher-concurrency configurations. This is consistent with the increasing amount of concurrent write activity, where several clients repeatedly attempt to replace the same DSM snapshot key. The larger error bars for the higher-concurrency cases also indicate greater variability across the three runs.

As with the read-latency chart, these measurements are end-to-end latencies observed by the browser clients. They include the cost of DSM communication, coverability checks, browser execution, Selenium overhead, and local contention from running multiple browser instances

on the same benchmark machine.

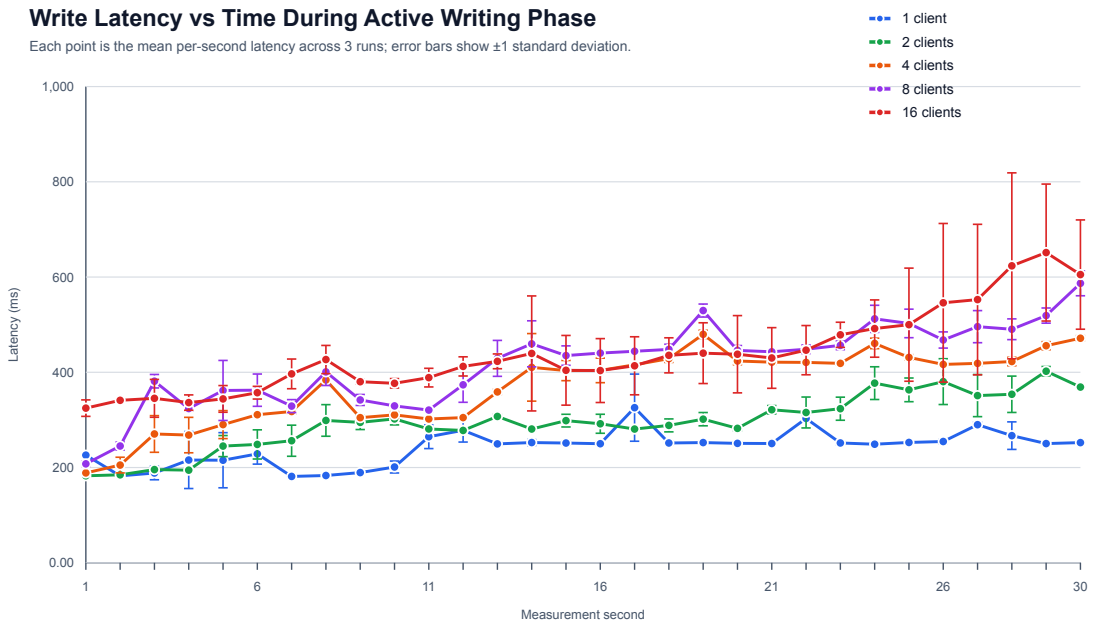


Figure 5.4: Write latency during the active writing phase.

5.6.6 Read Latency Percentiles vs Concurrent Clients

Figure 5.5 summarizes read latency using the p50, p95, and p99 percentiles for each client count. The p50 line represents the median read latency, while p95 and p99 show the tail latency experienced by the slower read operations.

The median read latency increases gradually, from approximately 250 ms with one client to approximately 400 ms with sixteen clients. The tail latency grows much more sharply. In particular, p99 read latency rises from approximately 400 ms with one client to more than 2.2 seconds with sixteen clients. This indicates that although typical read operations remain within a few hundred milliseconds, higher concurrency creates occasional much slower reads.

This behaviour is consistent with the read-latency-over-time results in Figure 5.3. As more browser clients poll and synchronize concurrently, the system experiences higher end-to-end delay and greater variability. Since the benchmark measures latency at the browser clients, these percentiles include DSM communication as well as local browser, Selenium, and host-machine scheduling overhead.

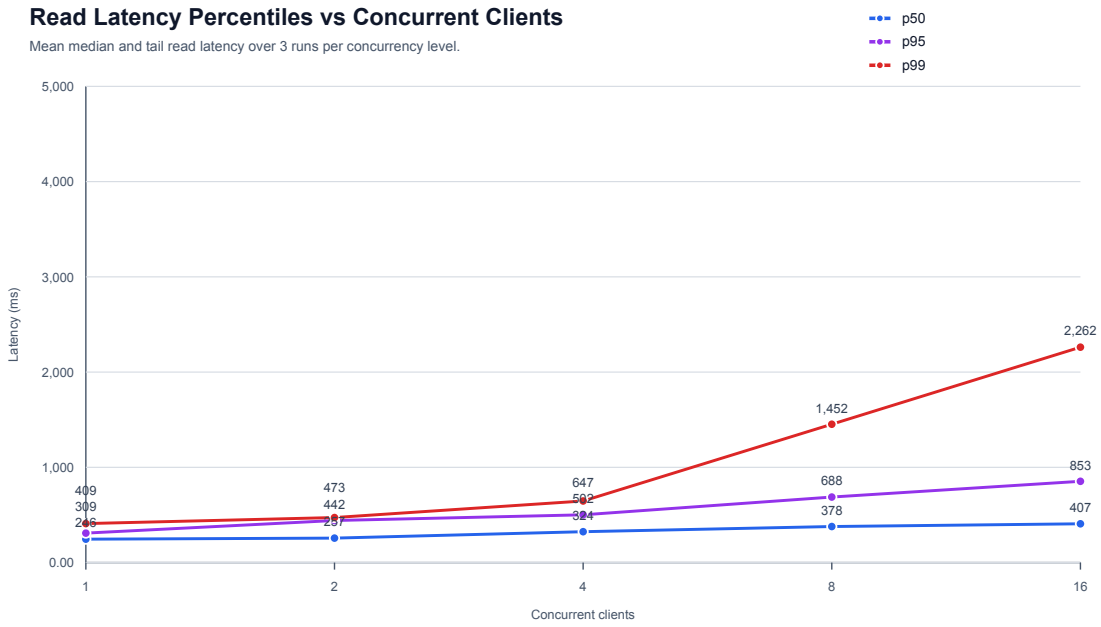


Figure 5.5: Read latency percentiles as the number of concurrent clients increases.

5.6.7 Write Latency Percentiles vs Concurrent Clients

Figure 5.6 summarizes write latency using the p50, p95, and p99 percentiles for each client count. These values refer to individual DSM write attempts, including both `SUCCESS` and `UNSUCCESS` responses. They therefore measure the latency of a single write request, not the total time required for a client to eventually succeed after one or more retries.

The chart shows that write latency increases steadily as the number of concurrent clients grows. The median write latency rises from approximately 250 ms with one client to approximately 390 ms with sixteen clients. Tail latency also increases: p95 rises from approximately 300 ms to 650 ms, while p99 rises from approximately 410 ms to 860 ms.

This behaviour is consistent with the write outcome results in Figure 5.2. As more clients concurrently attempt to replace the same DSM snapshot key, more write attempts encounter coverability conflicts and the system experiences higher end-to-end write latency. As with the other latency measurements, these values include both DSM communication and client-side benchmark overhead.

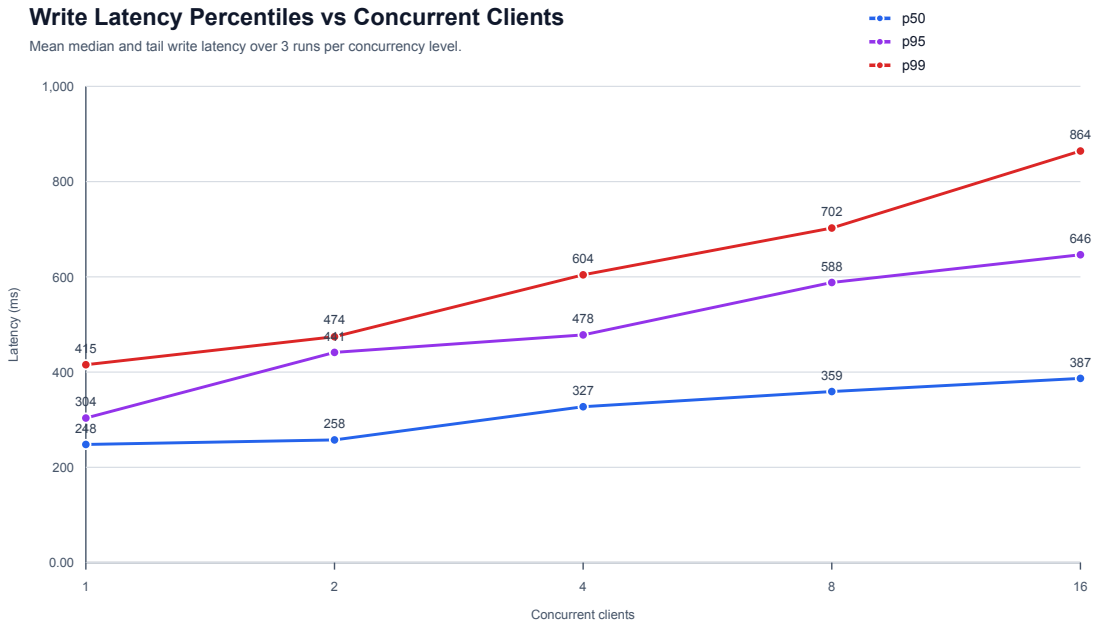


Figure 5.6: Write latency percentiles as the number of concurrent clients increases.

5.7 Comparison with the Conventional `y-websocket` Architecture

The results presented so far evaluate the proposed DSM-based editor using metrics that are specific to its synchronization algorithm, such as DSM read throughput, DSM write throughput, read and write latency, and write outcome composition. These metrics are useful for understanding the behaviour of the DSM-based design, but they do not have a direct equivalent in a conventional `y-websocket` architecture.

For this reason, a second experiment was performed using shared metrics that apply to both systems: primarily end-to-end token propagation latency and convergence time.

5.7.1 Comparison Setup and Propagation Latency Metric

The experiment used the same token-based workload as the main benchmark. Each client attempted to insert one uniquely identifiable token every 10 ms during a 30-second measurement period, after a 3-second warm-up period. The one-client case was omitted, since propagation to another client is not defined when there is no remote observer. The tested configurations therefore used 2, 4, 8, and 16 concurrent clients, and each configuration was repeated three times.

For the DSM-based system in this comparison experiment, the polling and synchronization loops were configured with a 200 ms interval. This matches the interactive demonstration configuration and represents a less aggressive client-side synchronization setting than the 1 ms stress-test

configuration used in the DSM-specific benchmark. The conventional `y-websocket` system does not have equivalent polling and synchronization-loop parameters, since updates are pushed through the WebSocket synchronization server.

For each token produced by a client, the benchmark recorded the time at which the token was inserted by the producer. At the other clients, the benchmark recorded the first time the same token was observed in the shared document. For a token T, produced by client P and observed by another client O, the propagation latency is defined as:

$$latency(P \to O, T) = firstSeen_O(T) - created_P(T).$$

Since all Selenium-driven browser instances ran on the same benchmark machine, these timestamps were taken from the same system clock and could therefore be compared directly.

A run with N clients produces one latency sample for each producer–observer–token combination. Thus, if a run produces T tokens in total, it yields $T \times (N - 1)$ propagation-latency samples. The reported values are computed over the samples collected from the three runs of each configuration.

The DSM-based version used the same prototype, the same four coverable DSM nodes, and the same four local application instances as the main experiment. Browser clients were again assigned to the application instances and to the DSM nodes in round-robin fashion. The baseline used the conventional `y-websocket` synchronization architecture, in which clients exchange Yjs updates through a WebSocket synchronization server. Since the DSM nodes used in the experiments were hosted on CloudLab, the WebSocket server was also hosted on CloudLab for a fairer comparison, specifically on an Emulab `d430` machine. The `y-websocket` system experiment used the same four application instances and the same round-robin browser assignment. In both cases, the same Selenium-based benchmark structure, workload, and correctness validation method were used.

Both systems passed the correctness checks in all tested configurations. In every run, all clients eventually converged to the same final document text, and every generated token appeared exactly once, with no missing, duplicate, or extra tokens.

5.7.2 Propagation Latency Comparison

Table 5.2 summarizes the end-to-end token propagation latency results. The values are reported in milliseconds.

The `y-websocket` system achieved lower propagation latency than the DSM-based system in all tested configurations. With two clients, the mean propagation latency was approximately 201 ms for the `y-websocket` system, compared with approximately 2021 ms for the DSM-based

Table 5.2: End-to-end propagation latency for the DSM-based system and the conventional `y-websocket` system. Values are in milliseconds and represent the mean over three runs.

Clients	System	Mean	p50	p95	p99
2	DSM	2021	1654	5085	6569
	y-websocket	201	197	213	349
4	DSM	3791	3157	9552	12340
	y-websocket	224	214	280	440
8	DSM	5569	5011	11942	15247
	y-websocket	304	274	523	680
16	DSM	10196	8855	22735	29167
	y-websocket	796	537	2328	3462

system, meaning that DSM latency was about 10.1 times higher. At four clients, the DSM mean latency was about 16.9 times higher, at eight clients, about 18.3 times higher, and at sixteen clients, about 12.8 times higher.

The tail latency shows the same pattern. For the DSM-based system, p99 propagation latency increased from approximately 6.6 seconds with two clients to approximately 29.2 seconds with sixteen clients, a 4.4x increase. For the `y-websocket` system, p99 latency increased from approximately 349 ms to approximately 3.46 seconds over the same range. Comparing the two systems directly, the DSM-based p99 latency was about 18.8 times higher than `y-websocket` at two clients, 28.0 times higher at four clients, 22.4 times higher at eight clients, and 8.4 times higher at sixteen clients. This indicates that, under the evaluated comparison workload, the DSM-based architecture experiences substantially higher user-visible propagation delay than the conventional `y-websocket` approach.

This result is expected from the design of the two systems. In the `y-websocket` system architecture, Yjs updates are pushed directly through the synchronization server to the other connected clients. In the DSM-based architecture, document changes become visible through periodic reads of the DSM snapshot, client-side merging, full-snapshot writes, and retries when coverability rejects a write based on an outdated snapshot. Therefore, although the DSM-based system removes the dedicated Yjs-aware synchronization server, it pays for this design choice with higher propagation latency.

5.7.3 Convergence Time Comparison

Figure 5.7 depicts the convergence time after the token-generating bots stopped. For the DSM-based system, convergence time increased from approximately 2.03 seconds with two clients to approximately 10.6 seconds with sixteen clients, corresponding to about a 5.2x increase. In contrast, the `y-websocket` system remained close to one second across the tested configurations, ranging from approximately 0.91 to 1.41 seconds.

The difference between the two systems also increases with concurrency. With two clients, the DSM-based system required about 2.0x more time to converge than the `y-websocket` system. With sixteen clients, this gap increased to approximately 7.5x. This indicates that, after active editing stops, the DSM-based system still needs additional read, merge, and write/retry cycles before all clients observe the same final document state.

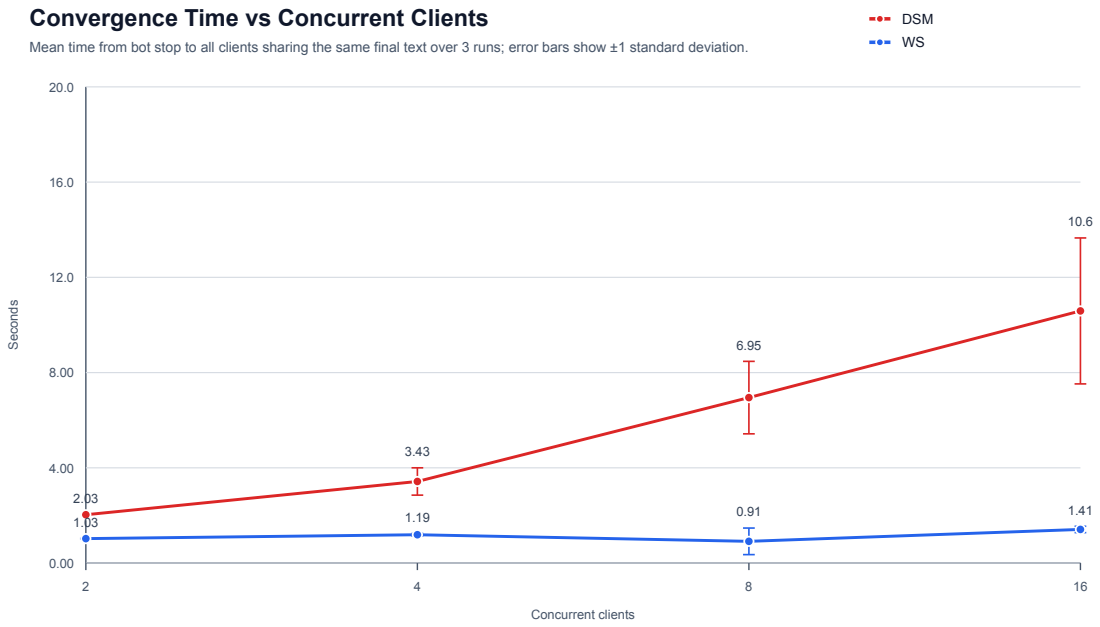


Figure 5.7: Time from bot stop until all clients share the same final text. Error bars show ± 1 standard deviation over three runs.

Overall, this comparison confirms the main trade-off of the proposed architecture. The conventional `y-websocket` system provides much lower propagation latency and faster convergence under the evaluated workload. At sixteen clients, the DSM-based system has mean propagation latency about 12.8x higher and convergence time about 7.5x higher than the `y-websocket` system. The DSM-based system, however, replaces the dedicated Yjs-aware synchronization server with a generic replicated storage backend, while still preserving correctness in all tested configurations.

5.8 Limitations

The main limitation of the evaluation is that all Selenium-controlled browser clients were executed on the same machine. As the number of clients increases, the benchmark host must run more browser instances, more JavaScript synchronization loops, and more Selenium interactions concurrently. Therefore, the measured latencies include not only DSM communication

and server-side processing, but also client-side CPU contention, browser scheduling overhead, and Selenium overhead.

A more representative scalability experiment would distribute the browser clients across multiple machines. For example, in the sixteen-client case, four CloudLab nodes could each run four Selenium browser instances, while still connecting to the same DSM backend. This would reduce the single-machine bottleneck and make the measurements better reflect the behaviour of the DSM-based synchronization architecture under distributed client load.

Such a setup would require coordination between the client machines so that the benchmark phases begin at approximately the same time. This could be implemented using a central controller that starts all workers, waits until every worker reports that its browser clients are ready, and then sends a shared start signal for the measurement phase. The results from all workers could then be collected and aggregated after the convergence checks complete.

Chapter 6

Conclusion and Future Work

6.1 Conclusion 52

6.2 Limitations 53

6.3 Future Work 54

6.1 Conclusion

This thesis examined whether a real-time collaborative editor can be built without a dedicated CRDT-aware synchronization server, by using a coverable DSM backend as the shared storage layer. The implemented system used Yjs for client-side document convergence and stored the collaborative document as a full-state snapshot in the DSM. The main question was whether this design could preserve correctness under concurrent editing, and what performance cost would appear as the number of clients increased.

The experimental results show that the proposed approach was correct for all tested configurations. With 1, 2, 4, 8, and 16 concurrent clients, all runs eventually converged to the same final document state. In addition, every generated token appeared exactly once in the final document, with no missing, duplicate, or extra tokens. This confirms that, for the evaluated workload, the combination of Yjs merging at the clients and coverable DSM writes was sufficient to prevent lost updates and preserve convergence.

The results also highlight the main scalability trade-off of the design under a stress-test workload. As concurrency increased from one to sixteen clients, aggregate read throughput increased from approximately 4.19 to 32.5 operations per second, while aggregate write throughput increased from approximately 4.25 to 33.5 operations per second. This corresponds to about a 7.8x–7.9x throughput increase, even though the number of clients increased by 16x. This was expected because more clients produced more read and write operations, while also competing to update the same DSM snapshot key.

The write outcome results make this contention visible. With one client, all writes succeeded. As the number of concurrent clients increased, more write attempts were rejected as stale and had to

be retried. At sixteen clients, unsuccessful write attempts formed approximately 85% of all write attempts, meaning that rejected writes were substantially more frequent than successful writes. This confirms that the single-snapshot design preserves correctness, but creates increasing write contention under concurrent editing.

Latency followed the same general pattern. Both read and write latency increased as more clients were added, and tail latency became more pronounced at higher concurrency. For read operations, median latency increased by approximately 65% from one to sixteen clients, while p99 latency increased by more than 450%. Write latency increased more moderately, with p99 latency increasing by approximately 108% over the same range. These results indicate that the design remains functional under concurrent editing, but that repeated reads, full-snapshot writes, and coverability retries become increasingly visible when the system is placed under heavy concurrent load.

The comparison with the conventional `y-websocket` system further clarified the trade-off introduced by the proposed architecture. The `y-websocket` system achieved substantially lower propagation latency and faster convergence under the evaluated workload. Across the tested configurations, the DSM-based system had mean propagation latency approximately 10x to 18x higher than `y-websocket`. At sixteen clients, its convergence time was also about 7.5x higher. These results show that the DSM-based system preserves correctness while replacing the dedicated Yjs-aware synchronization server, but it does so with a clear performance cost in user-visible propagation latency and convergence time.

Overall, the thesis demonstrates that a DSM-backed collaborative editor is feasible and can preserve correctness without a central Yjs-aware synchronization server. The prototype works as a collaborative editor and successfully maintains convergence in all evaluated configurations. At the same time, the stress-test evaluation reveals the expected performance trade-offs of this design, mainly due to polling, full-snapshot writes, and coverability retries under heavy concurrent editing.

6.2 Limitations

The main limitation of the evaluation is that all Selenium browser clients were executed on the same machine. Therefore, the measured latencies include not only DSM communication and synchronization cost, but also browser scheduling, Selenium overhead, CPU contention, and local host limitations. This is especially relevant for the 8-client and 16-client experiments, where the benchmark machine itself may have become a bottleneck.

A second limitation is the scale of the experiments. The system was tested with up to sixteen concurrent clients and three runs per configuration. This was enough to demonstrate correctness

and reveal the first performance bottlenecks, but it is not sufficient to make general claims about large-scale deployments.

The workload was also simplified. Each client repeatedly inserted uniquely identifiable tokens into the document. This was useful for correctness validation, because the final document could be checked precisely. However, real collaborative editing includes pauses, deletions, formatting changes, larger documents, and uneven editing patterns. These cases were not fully explored in the current evaluation.

Moreover, the benchmark intentionally created a high-contention workload by making all clients insert tokens frequently into the same shared document. This is useful for stress-testing the synchronization design, but it is more aggressive than typical human collaborative editing behaviour.

Finally, the experiments did not test DSM node failures, network delays, or reconfiguration. These would be necessary to evaluate the practical advantages and disadvantages of the DSM-based architecture more completely under distributed and failure-prone conditions.

6.3 Future Work

A first direction for future work is to repeat the experiments in a distributed benchmark environment. Instead of running all Selenium instances on one machine, clients could be spread across multiple CloudLab nodes. For example, four machines could each run four browser clients for the sixteen-client case. This would reduce the local benchmark bottleneck and make the measured results more representative of a distributed deployment. Such an experiment would require coordination between the machines so that all clients start the warm-up, measurement, and convergence phases at approximately the same time.

A second direction is to reduce write contention. The current design stores the whole document snapshot under one DSM key, so all writers compete for the same object. Future versions could add random jitter, adaptive synchronization intervals, or backoff after rejected writes. These changes could reduce repeated collisions between clients and lower the retry rate under high concurrency.

Future work should also evaluate more realistic editing workloads. This includes mixed insertions and deletions, rich-text formatting, larger documents, late-joining clients, and bursty user behaviour. It would also be useful to measure how snapshot size grows over time, since the current design repeatedly writes full-state Yjs snapshots.

Together, these directions would provide a more complete understanding of the scalability and practicality of DSM-backed collaborative editing systems under realistic deployment conditions, while building on the correctness and feasibility demonstrated by the current prototype.

BIBLIOGRAPHY

[1] Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating CRDTs for real-time document editing. In *Proceedings of the 11th ACM Symposium on Document Engineering*, DocEng '11, pages 103–112. Association for Computing Machinery, 2011.

[2] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[3] Clarence A. Ellis and Simon J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407. Association for Computing Machinery, 1989.

[4] Antonio Fernández Anta, Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, Efstathios Stavrakis, and Andria Trigeorgi. Boosting concurrency and fault-tolerance for reconfigurable shared large object. Under submission, 2024.

[5] Joseph Gentle and Martin Kleppmann. Collaborative text editing with eg-walker: Better, faster, smaller. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 682–700. Association for Computing Machinery, 2025.

[6] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[7] Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[8] Nancy A. Lynch and Alex A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing*, pages 272–281. IEEE, 1997.

[9] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 19th International Conference on Supporting Group Work*, GROUP '16, pages 39–49. Association for Computing Machinery, 2016.

[10] Nicolas Nicolaou, Viveck Cadambe, Narayana Prakash, Andria Trigeorgi, Kishori Konwar, Muriel Médard, and Nancy Lynch. ARES: Adaptive, reconfigurable, erasure coded, atomic storage. *ACM Transactions on Storage*, 18(4):1–39, 2022.

[11] Nicolas Nicolaou, Antonio Fernández Anta, and Chryssis Georgiou. Cover-ability: Consistent versioning in asynchronous, fail-prone, message-passing environments. In *2016 IEEE 15th International Symposium on Network Computing and Applications*, pages 224–231. IEEE, 2016.

[12] Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403. IEEE, 2009.

[13] Quill. Quill rich text editor. https://quilljs.com/.

[14] Selenium. Webdriver. https://www.selenium.dev/documentation/webdriver/.

[15] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011.

[16] Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68. Association for Computing Machinery, 1998.

[17] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.

[18] Yjs. A collaborative editor. https://docs.yjs.dev/getting-started/a-collaborative-editor.

[19] Yjs. Document updates. https://docs.yjs.dev/api/document-updates.

[20] Yjs. Introduction. https://docs.yjs.dev/.

[21] Yjs. y-quill: Quill editor binding for yjs. https://github.com/yjs/y-quill.

[22] Yjs. y-websocket. https://docs.yjs.dev/ecosystem/connection-provider/y-websocket.

[23] Yjs. Yjs encoding protocols. https://github.com/yjs/y-protocols/blob/master/PROTOCOL.md.

[24] Yjs. Yjs internals. https://github.com/yjs/yjs/blob/main/INTERNALS.md.