A Paxos-Based Strongly Consistent Live Document Editor

WEIXIN YU* and KAYLEE XIE*, Stanford University, USA

Collaborative document editors such as Google Docs and Visual Studio Code Live Share achieve real-time editing through Operational Transformation (OT) or Conflict-free Replicated Data Types (CRDTs), prioritizing availability at the cost of strong consistency. Conflicting concurrent edits must be merged, introducing subtle ordering anomalies and significant implementation complexity. In this project, we present a strongly consistent distributed document editor built on the Paxos consensus protocol [5], implemented in C++ using the coroutine facilities introduced in C++20. By serializing all document operations through a consensus log, our system guarantees that every participating process observes an identical, conflict-free edit history. We implement a custom cooperative coroutine runtime and awaitable message queue abstraction that express the Paxos protocol phases—leader election, log replication, and crash recovery as straightforward sequential coroutines without callbacks or explicit state machines. We evaluate the system on clusters of up to 55 nodes. Cold leader election completes in under 175 ms across all cluster sizes with zero failures. End-to-end mean write latency from client submission to all-alive-followers commit ranges from 197.5 ms at 3 nodes to 227.98 ms at 55 nodes (all running in separate processes on a single machine), and is insensitive to payload size across three orders of magnitude. After a leader crash, a new leader is elected within 125 ms with 100% reliability. Our results demonstrate that strong consistency is achievable at practical latencies for a document editing workload.

1 Introduction

Real-time collaborative document editing is a widely used distributed-systems application. Systems such as Google Docs and Visual Studio Code Live Share allow multiple users to edit a shared document simultaneously, with changes propagated to all participants in near-real time. Mainstream systems achieve responsiveness through Operational Transformation (OT) [3] or Conflict-free Replicated Data Types (CRDTs) [8], two frameworks that permit concurrent edits to be applied in any order and converge eventually. While providing causality and low latency in practice, both approaches provides only **eventual consistency**: there is no single authoritative order of operations, and the merge algorithms required to reconcile concurrent edits are notoriously difficult to implement correctly. These systems create the illusion that users will always see exactly what they typed, but arbitrary message ordering can produce different outcomes. In addition, OT systems such as Google Docs typically rely on a **central fault-tolerant** server to serialize updates, whereas CRDT-based systems often allow replicas to accept local updates before synchronization. As a result, a locally generated update may be **lost** if the originating node crashes permanently before that update is durably replicated elsewhere.

We take a different approach. Instead of allowing divergent replicas and later reconciling them, we serialize every document edit through leader-based Paxos [5] [6] [9] so that all nodes commit the same ordered sequence of operations, and will **never lose committed update** as long as a majority of the nodes is active. In the collaborative editing setting, linearizing all operations across participants and apply them in log order simply avoids the problem of dealing with human intention and causality with rule-based systems. This **strong ordering** eliminates ambiguity and prevents conflicts from any concurrent edits, at the cost of additional coordination latency. Applications of collaborative editors, such as sign-up sheets, can also benefit from the strict ordering. Since the leader is just a performance optimization, it can be dynamically configured, and so **no centralized server** is necessary to provide the service. In this project, we evaluate the latency of paxos-based live editor both quantitatively and qualitatively.

A key design choice is the use of C++20 coroutines to implement the Paxos state machine. Consensus protocols involve many concurrent, interleaved activities—leader election, heartbeat monitoring, per-slot proposal tracking, and log recovery—each of which must suspend until network responses or timeouts are received. Traditional implementations express these as explicit finite-state machines distributed across callbacks or threads, which are difficult to reason about and debug. C++20 stackless coroutines allow us to express each activity as a sequential loop that *co_awaits* network messages or timers, yielding code that closely mirrors the protocol specification. Real-world consensus implementations [1] confirm that bridging the gap between protocol description and production code is a central engineering challenge; our coroutine-based design directly addresses this.

The highlights of this work are:

(1) A C++20 coroutine runtime comprising a cooperative tick-based scheduler, an awaitable message queue (Mailbox<T>), and a timer awaitable (SleepFor).
(2) A complete Multi-Paxos implementation expressed as three concurrent coroutines (message loop, election loop, heartbeat loop) per node, handling leader election, log replication, request forwarding, and crash recovery via a synchronization protocol.

*Both authors contributed equally to this project.

Authors' Contact Information: Weixin Yu, weixinyu@stanford.edu; Kaylee Xie, mengjiex@stanford.edu, Stanford University, Stanford, CA, USA.

(3) A TCP transport layer that bridges a dedicated I/O thread to the coroutine runtime via a mutex-protected handoff.
(4) A collaborative text editor with a browser-based frontend that supports arbitrary insertion, deletion, and copy-pasting with latency imperceptible to users in noraml operating conditions.
(5) An empirical evaluation on clusters of up to 55 nodes showing that cold leader election completes in under 175 ms, end-to-end write latency (client submit to all alive followers commit) increases from 197.5 ms to 228.0 ms across an 18× increase in cluster size, payload size has negligible effect on latency, and crash recovery completes within 125 ms with zero failures at a well-chosen election timeout.

2 Background

2.1 Groupware Systems, OT, and CRDTs

Groupware systems support two or more users engaged in a common task. Real-time group editors are one of the important applications, with which multiple users can edit a shared document simultaneously. Ensuring consistency across multiple replicas in these systems is a central challenge because users may perform conflicting concurrent operations. Operational Transformation (OT) [3] was one of the earliest approaches developed to address this problem. In OT, each user's operation is transformed against other users' update by a central authority, so that all replicas converge to the same document state while preserving the original intentions of each edit. Google Wave [4], which was later used in Google Docs, is based on OT. In OT, the centralized server is a **single point of failure**.

More recently, Conflict-free Replicated Data Type (CRDT) [8] enables peer-to-peer style distributed and decentralized replication. It uses data structures whose operations naturally commute, allowing replicas to converge without requiring a central transformation engine. CRDTs underlie systems built with libraries like Yjs, including Evernote and JupyterLab [2]. However, if a node permanently **loses data** that isn't already propagated, the operations may be lost.

A collaborative editor using OT or CRDT may produce non-intuitive results under concurrent edits. Consider the document below. Suppose two clients become partially partitioned. Client A *deletes paragraph 2*, while Client B *moves paragraph 2 to after paragraph 3*. After synchronization, one possible merged result is shown on the right. Although the system converges, the final state may not match either users' intent.

```
Initial:
paragraph 1
paragraph 2
paragraph 3
```

```
Result:
paragraph 1 paragraph 3
paragraph 2
```

Both OT and CRDT prioritize low-latency collaboration over strict global ordering, which can yield results that diverge from user intent. Because they provide only **eventual consistency**, this tradeoff motivates us to explore Paxos as a consensus-based approach to achieve strong consistency in collaborative editing.

2.2 Consensus Protocols

Consesnsus-based protocols like Paxos [5] [6] [9] and Raft [7] guarantee **strong consistency** to ensure that all nodes agree on a single sequence of operations. The protocol is also inherently **fault-tolerant** to up to $\lfloor N/2 \rfloor$ node failures in a cluster of N nodes. Each operation is proposed and accepted through a majority agreement, ensuring any committed operations are never lost. Leader-based variants further optimize throughput by skipping the *prepare* phase in normal operating conditions, where the leader have access to a majority of follower nodes.

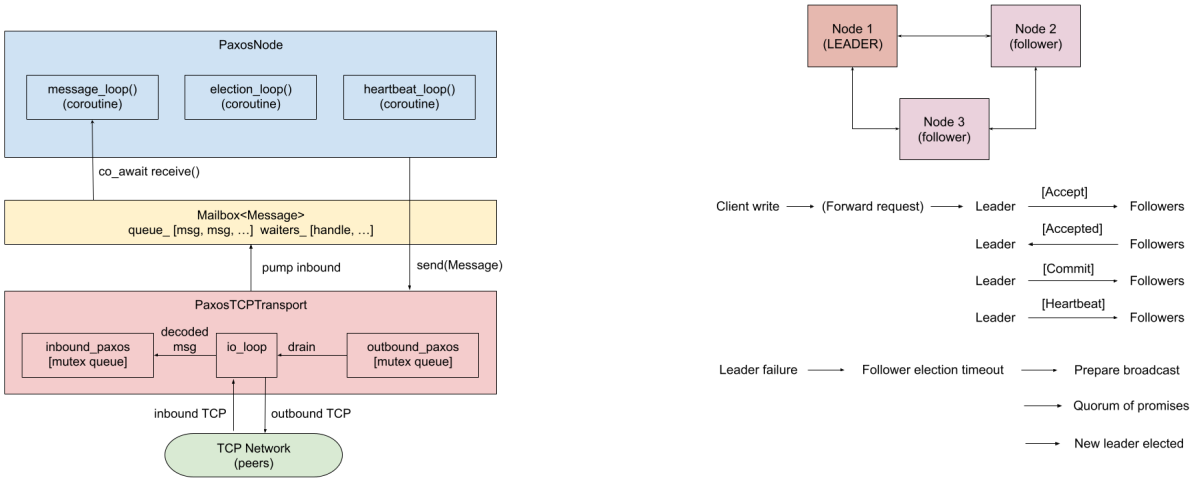
2.3 C++20 Coroutines

C++20 coroutines allow functions to suspend and resume execution using *co_await*, enabling cooperative multitasking without requiring one thread per task. Building Paxos with coroutines provides a lightweight and readable implementation of a replicated state machine, hiding stack-ripping and callback complexity.

3 System Overview

We call every machine, or client, that runs the system, a node. The collection of all nodes connected together via the TCP transport layer is called a cluster. Right now we don't support dynamic node configuration, so all ip addresses and ports are known before the start of any one node. This is a design decision made out of expediency.

Each node in the cluster runs three concurrent layers: a coroutine runtime that drives all Paxos logic, a PaxosNode that implements the Multi-Paxos state machine as coroutines, and a TCP transport that runs a dedicated I/O thread and bridges incoming bytes into the coroutine runtime. Figure 1a shows the per-node structure; Figure 1b shows a three-node cluster.



(a) Per-Node Architecture

(b) Three-Node Cluster ($n = 3$, quorum=2)

Fig. 1. System architecture and example cluster layout.

3.1 Paxos Design

The system adopts leader-based Paxos to reduce consensus overhead. In this report, we refer to non-leader replicas as follower nodes. The leader periodically broadcasts heartbeats to the followers, and any follower may initiate a leader election after its election timeout expires. These timeouts are intentionally chosen to differ across nodes in order to reduce simultaneous elections and avoid livelock.

Prepare Phase: At cluster startup, or whenever a node does not receive a leader heartbeat within its election timeout, it initiates the prepare phase of leader election. The candidate proposes a globally unique ballot and becomes leader after receiving `Promise` responses from a quorum of nodes. A node returns a `Promise` only if the ballot is the highest it has observed, and includes any previously accepted log values in its reply.

Accept Phase: After leadership is established, the leader proposes client commands for successive log slots without repeating the prepare phase. Once it receives `Accepted` responses from a quorum, the entry is safe to commit and the leader broadcasts a `Commit`. Followers that miss the commit due to message loss, temporary failure, or a network partition later detect the gap through the commit index piggybacked on leader heartbeats and recover via a `sync_request-sync_data` exchange.

During normal operation, when the leader is reachable by a majority, a command commits in 1.5 RTTs after the leader receives the client request: 1 RTT to collect a quorum of `Accepted` replies and 0.5 RTT to disseminate the commit. If the client is on a non-leader node, an additional 0.5 RTT is needed to forward the request to the leader.

4 Implementation

4.1 Coroutine Runtime

The coroutine runtime (`runtime.hpp`) is a lightweight cooperative scheduler that executes all protocol logic on a single thread. It maintains a queue of runnable coroutines and a timer structure for delayed wakeups, advancing logical time only when no coroutine is immediately runnable. This design is sufficient for our event-driven Paxos implementation, where timing is needed primarily for leader heartbeats and election timeouts rather than for real-time scheduling.

`Mailbox<T>` (`nos.hpp`) provides the basic asynchronous communication primitive between coroutines. It supports message passing within the runtime by allowing coroutines to either consume available messages immediately or suspend until a new message arrives. This abstraction lets protocol components communicate through sequential coroutine code rather than callbacks or explicit state machines.

`SleepFor` is a timer awaitable that suspends a coroutine for a specified number of logical ticks, enabling periodic tasks such as heartbeat transmission and election timeout checking.

4.2 Multi-Paxos Implementation

`PaxosNode` (`paxos.hpp`, `paxos.cpp`) implements Multi-Paxos as three coroutines running on a shared single-threaded `Runtime` and communicating through a common `Mailbox<Message>` inbox. This design avoids explicit locking while allowing the protocol to be expressed as sequential coroutine logic.

The first coroutine processes incoming messages and dispatches them to the appropriate Paxos handlers. The second manages leader election by periodically checking whether the node has exceeded its election timeout without hearing from a leader, in which case it begins a new election with a globally unique ballot number. The third sends periodic heartbeats when the node is the leader, allowing followers to detect leader liveness and identify missing committed log entries.

Together with a local `submit_command` interface for client requests, these coroutines implement the protocol described in Section 3.1, including leader election, log replication, and follower catch-up through explicit synchronization messages.

4.3 TCP Transport

`PaxosTcpTransport` (`paxos_tcp_transport.hpp`, `src/paxos_tcp_transport.cpp`) separates blocking TCP communication from the single-threaded coroutine runtime. A dedicated I/O thread manages network connections and message transmission, while the main thread periodically transfers received messages into the corresponding Paxos node's mailbox. This design preserves the runtime's single-threaded execution model and avoids locking within the Paxos protocol logic.

The transport also provides fault-injection support through `crash()` and `restore()`, which simulate temporary node failure and recovery. While crashed, a node neither sends protocol messages nor participates in leader election; once restored, it resumes normal communication and can rejoin the cluster through the standard synchronization mechanism.

4.4 Collaborative Editor Application

4.4.1 Overview. On top of the leader-based Paxos implementation that accepts client commands as raw C++ strings, we build the collaborative editor application. Users interact with a text area similar to those commonly seen on web pages, as shown in Figure 3. As they type, changes are applied only after the corresponding operation is committed by the underlying Paxos cluster. This guarantees strong consistency and eliminates divergence between what the user expects and the eventual state of the document. Changes made by other users will also appear in commit order on the screen.

As shown in Figure 2, the users will be served at a browser page that connects to the local Paxos node binary with bidirectional message, with Node.js as a bridge. We use websocket to communicate between the browser frontend and the TCP server. Unlike http, the connection with web socket stays open. This enables low-latency edits and server-program message pushing. For the connection between Node.js and the Paxos node, we use raw TCP connection, where the paxos node servers as a server. On an update is made by the browser, the command is propagated to the Paxos node, put in the `input_queue`, which the main thread polls. Once a Paxos node knows a new slot in Paxos's replicated `committed_log` is committed, it propagates the command to the browser frontend.

4.4.2 Application-layer Retry Mechanism. Because the Paxos interface accepts raw string commands, client requests may be dropped in a network that tolerates arbitrary message loss. While safety and consistency are preserved—uncommitted updates never appear in the browser—dropped requests may prevent progress. To ensure liveness, the client retries timed-out requests and sends them to different nodes in the cluster in a round-robin fashion.

To support deduplication, each command sent to the Paxos replicated state machine includes a `client_id` and a `request_id`. The system uses these identifiers to prevent duplicate requests from being appended to the log.

5 Evaluation

We evaluate three aspects of our Paxos implementation: cold-start leader election latency, end-to-end write latency, and leader crash recovery time. All experiments run on a single machine with all nodes as local processes communicating over loopback TCP, so network latency is negligible and measurements reflect protocol overhead alone. The runtime's logical clock advances in 5 ms ticks; each node polls for inbound messages on every tick.

5.1 Cold Leader Election Latency

Setup. We start n nodes simultaneously with no pre-existing leader. We measure the time from the first node's startup to the moment any node emits a `leader_elected` trace, across cluster sizes $n \in \{3, 5, 11, 23, 55\}$ with 30 trials each (150 trials total).

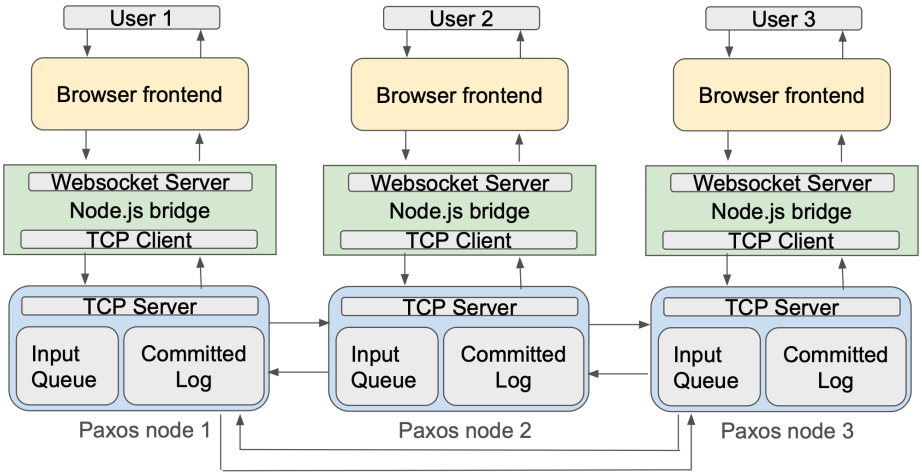


Fig. 2. Message flow of the collaborative editor application in a 3 nodes cluster.

Fig. 3. The frontend of the collaborative editor application.

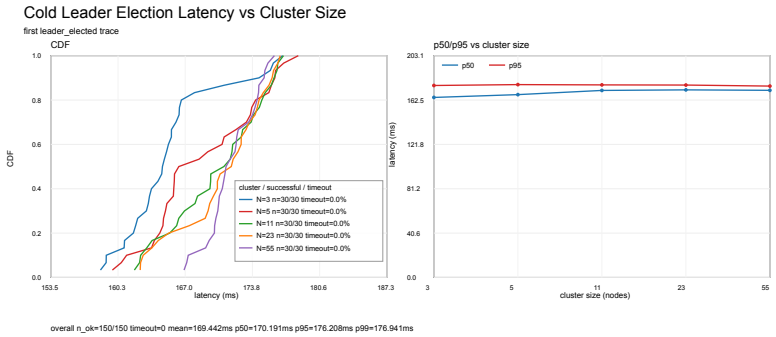


Fig. 4

Results. All 150 trials elected a leader successfully. As shown in Figure 4, election latency is nearly independent of cluster size: mean latency grows from 165.5 ms at $n = 3$ to only 171.7 ms at $n = 55$, a 4% increase over an 18× scale-up. The p95 latency stays within a 1 ms band across all cluster sizes (175–177 ms).

Discussion. This near-constant scaling is expected. The election protocol requires only one quorum of `Promise` replies, so the critical path is one round-trip latency plus the election timeout, both of which are independent of n. The small increase at larger cluster sizes reflects slightly higher contention as more nodes simultaneously begin their own election timers.

5.2 End-to-End Write Latency

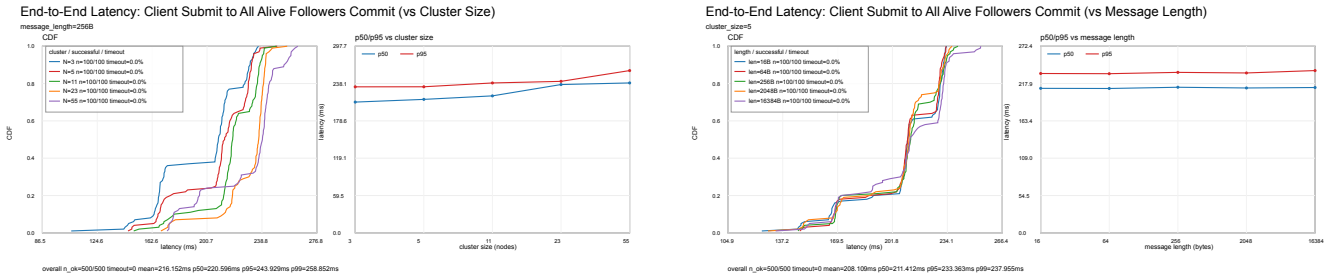
Setup. A client submits requests in round-robin fashion across all nodes. We measure the time from submission until all alive follower nodes have reflected the commit in their log. We run 100 samples per configuration.

Cluster size sweep (Figure 5a). Holding message size fixed at 256 bytes and sweeping $n \in \{3, 5, 11, 23, 55\}$, mean latency is 197.5 ms, 208.2 ms, 218.0 ms, 229.0 ms, and 228.0 ms, respectively. All 500 samples succeed.

Message size sweep (Figure 5b). Holding $n = 5$ fixed and sweeping payload size from 16 B to 16 KB, mean latency remains nearly flat: 208.4 ms (16 B), 208.3 ms (64 B), 208.1 ms (256 B), 206.4 ms (2 KB), and 209.4 ms (16 KB), with all 500 samples succeeding.

Discussion. Across both sweeps, end-to-end write latency remains practical but reflects our stricter completion criterion: we measure until *all alive followers* observe the commit, not just a quorum. With 256 B messages, mean latency rises from 197.5 ms at $n = 3$ to about 228 ms at $n = 55$, and p95 increases from 232.9 ms to 258.9 ms, consistent with larger-cluster straggler effects when waiting for every follower. In contrast, at fixed $n = 5$, latency is largely insensitive to payload size from 16 B to 16 KB (means 206.4–209.4 ms), indicating coordination and scheduling overhead dominate serialization cost on loopback TCP. All 1,000 samples in these two sweeps succeeded (0% timeout/failure), showing stable behavior under this all-alive-follower metric.

- Weixin Yu and Kaylee Xie



(a) Cluster-size sweep

(b) Message-size sweep

Fig. 5. End-to-end write latency under cluster-size and payload-size sweeps.

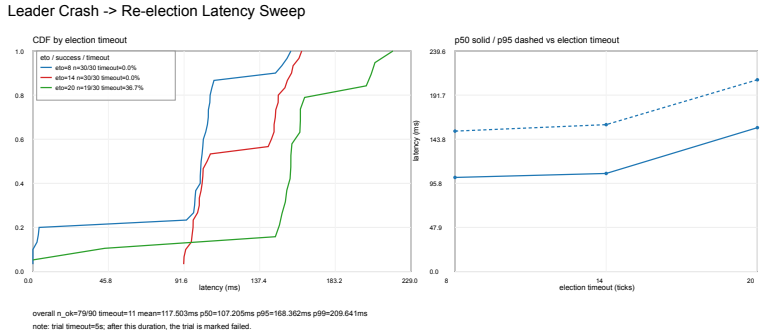


Fig. 6. Leader crash and re-election latency under different election timeouts.

5.3 Leader Crash and Re-election

Setup. In a 5-node cluster, we simulate a leader crash after the system has reached a stable state. We then measure the time from the crash event to the election of a new leader, sweeping the election-timeout base $B \in 8, 14, 20$ ticks (40, 70, and 100,ms, respectively). The heartbeat interval remains fixed at 2 ticks (10,ms), and we run 30 trials for each setting.

Results. As shown in Figure 6, recovery time increases with the election timeout, as expected: at $B = 8$, all 30 trials succeed with a mean of 89.1 ms; at $B = 14$, all 30 trials succeed with a mean of 123.4 ms. At $B = 20$, only 19 of 30 trials succeed (36.7% failure rate) within the measurement window. The failed trials are not liveness failures; rather, they are measurement timeouts caused by recovery extending past the fixed observation window, which we deem too long to remain imperceptible to users.

Discussion. These results confirm a fundamental tradeoff in Paxos-style leader election: a shorter election timeout reduces recovery latency but increases the risk of false positives, whereas a longer timeout provides more safety margin at the cost of availability during genuine failures. With a heartbeat period of 2 ticks and a base timeout of 8 ticks, there are four heartbeat intervals before any node starts an election, which is sufficient to absorb transient delays. At $B = 20$, the gap is ten heartbeat intervals, making recovery slow enough to exceed the test window under unlucky jitter. For this deployment, $B = 14$ (70 ms base) represents a reasonable default, achieving zero failures and a mean recovery time below 125 ms.

6 Conclusion

We built a strongly consistent live document editor by placing document updates on top of a leader-based Paxos log implemented in C++20 coroutines. Our design replaces replica divergence and merge logic with a single committed order of edits, while a lightweight coroutine runtime makes the consensus protocol easier to express as sequential control flow.

7 AI Usage Disclosure

AI is used in this project to provide the initial starting point of paxos implementation, creating figures based on collected data, and polish the report.

References

[1] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '07)*. Association for Computing Machinery, New York, NY, USA, 398–407. doi:10.1145/1281100.1281103

[2] Yjs Contributors. 2026. Yjs – Shared data types for building collaborative software. https://github.com/yjs/yjs. Version tracked at commit, accessed on <DATE>.

[3] C. A. Ellis and S. J. Gibbs. 1989. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 399–407. doi:10.1145/67544.66963

[4] Andres Ferrate. 2010. *Google wave: up and running*. " O'Reilly Media, Inc.".

[5] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. doi:10.1145/279227.279229

[6] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[7] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[8] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://inria.hal.science/inria-00555588

[9] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 1–36.