LiteDoc: Make Collaborative Editing Fast, Scalable, and Robust

Saptaparni Kumar, Haochen Pan, Roger Wang, Lewis Tseng
Computer Science Department
Boston College
Chestnut Hill, MA, USA
{saptaparni.kumar, haochen.pan, wangbrh, lewis.tseng}@bc.edu

***Abstract*—Collaborative text editing applications like Google docs, Etherpad and Overleaf allow users to concurrently edit a "shared" document. Most existing collaborative text editing software require total ordering on the updates made to the document, which is achieved using a centralized sever or some form of consensus algorithm. Then on top of the ordering, the editor uses either operational transformation (OT) or differential synchronization (diff-sync) to apply the ordered update events to the already committed changes on their local copies. If there is no delay or failure, then eventually all updates can be applied correctly in the agreed order.**

Unfortunately, not only are these methods computationally intensive but they often result in conflicts due to users writing to the same location. It has also been proved that the metadata overhead for such protocols are at least linear in the number of delete events. Moreover, these event-based and diff-based algorithms are exceptionally difficult to implement and there are no provably correct solutions to these problems in the face of heavy concurrency. These collaborative editors either provide no proven guarantees or only provide eventual guarantees for correctness.

With LiteDoc, we propose a different approach to tackle this problem: we make collaborative editing fast, scalable and robust by providing simplified semantics. More importantly, we can formally prove that LiteDoc achieves deterministic guarantees of correctness. LiteDoc divides the shared document into several sections and allow *only* one user to write at a particular section at any given time. This removes all conflicts that arise from having multiple writers writing to the same location. This mechanism also obviates the task of implementing cumbersome modules for OT, diff-sync and rollbacks in case of conflicts. Note that while LiteDoc supports less features than general collaborative editors like Google docs, it is natural (and courteous) to avoid concurrent writing to the same location when multiple people collaborate.

***Index Terms*—Collaborative editor; System architecture; Correctness; Register; Atomicity; Asynchrony**

I. Introduction

With the advancement of the Internet, we are moving towards the always-connected working style. Collaborative text editing is one of the fundamental tools to enable remote collaboration; hence, it has been studied extensively in the literature, e.g., [1], [2]. There are several editors in market as well, such as Google docs [3], Etherpad [4], Overleaf [5], etc. To the best of our knowledge, the current implementation and design of such editors are centered around application design, i.e., most researches are focused on providing richer features, and treating underlying system architecture as a black box. As a result, there are several drawbacks due to the fundamental limitations of the underlying Interenet and distributed systems. In this paper, we advocate that in order to provide a fast, scalable, and robust collaborative editor, we have to start from the ground up. Towards this end, we present the design of LiteDoc, and argue why it achieves the three important properties. The key is that LiteDoc provides only simplified semantics (or features) so that these semantics can be satisfied using simple and theoretically proven distributed primitives such as atomic registers and writer-lock.

In Section II, we discuss limitations of the current implementation. We also provide a primitive simulation to demonstrate that current approach is fundamentally limited in performance due to the bottleneck in the underlying distributed consensus algorithms. In Section III, we present the design of LiteDoc, and briefly argue how we can formally prove its correctness. In Section IV, we present our preliminary design of the user interface. We present closely related work in Section V, and discuss exciting future work in Section VI.

II. Motivation

A. Existing Approaches

Collaborative editors available in the market like Google docs [3], Etherpad [4], Overleaf [5], etc, are event-driven in nature with extremely granular update events such as key press or mouse tap. A fundamental issue is to order the events so that clients observe "consistent" changes to the shared document. These application either use a single server to order these events or use a consensus algorithm among various servers to attain agreement on ordering of events. Once an ordering on the events has been determined, the application then updates the various local copies using consistency maintenance techniques like operation-transform (OT) and differential synchronization (diff-sync), etc. These techniques allow the new updates from the server to be applied on top of the user's changes that have already been committed on the editor/document in the local copy (i.e., the document saved on client's computer).

If all updates were commutative, applying these changes would be less of a hassle. This is one of the basic assumptions made by CRDTs (Conflict-Free Replicated Data Types) introduced by Shapiro et.al [6] and this make the problem of collaborative editing with commutative events relatively easier to handle. But languages, by nature, are not commutative and thus these softwares try to come up with clever mechanisms to perform OT and diff-sync to maintain data consistency.

Limitations: In spite of all the effort to build robust and scalable collaborative editors, we see that these editors are still faced with issues below:

- Local changes get lost if connection is lost or a conflict arises during OT or diff-sync.
- Gibberish text might appear because concurrent users tried to write at the same location and the consistency maintenance algorithm interleaved their requests in an unwanted manner.
- These editors suffer performance bottleneck due to single server or the underlying consensus algorithm. In the worst case, editors might not be available in the event of server failures or long delay (due to the famous FLP impossibility [7]).
- Worse, the task of implementing the consistency maintenance algorithms is notoriously difficult. Take for example operation transform (OT); Joseph Gentle, a former Google Wave engineer and an author of the Share.JS library wrote, "Unfortunately, implementing OT sucks. There's a million algorithms with different tradeoffs, mostly trapped in academic papers . . . Wave took 2 years to write and if we rewrote it today, it would take almost as long to write a second time."

B. Key Observation and High-level Design

We start with our key observation: Collaborative editors should support concurrency; however, it should *not* allow multiple clients to write to the same location at the same time, because this actually contradicts with how human collaborative with each other. In existing editors, if two or more users try to write to the same location in a document we can have any one of the following outcomes: (i) one user overwrites the others' text (ii) some (or all) users overwrite others' texts partially (iii) users get their texts interleaved resulting in gibberish text (currently in Google Docs). None of the scenarios is ideal.

Building on top of our key observation, we can simplify the design and implementation of a collaborative editor with a few reasonable trade-offs. In most cases, collaborative-editor users are smart enough to adapt their update mechanisms to avoid conflict. If we constrain the user behavior pattern in a certain way, we can achieve all three goals: speed, scalability, and robustness.

We take a radically different approach to tackling the problem of collaborative editing. We provide only limited semantics in our editor LiteDoc, and make sure these semantics can be supported (without using expensive protocols like consensus). In particularly, we divide the "shared" document into parts and give *temporary ownership* to users modifying each section. As a result we root out conflicts arising from two users trying to modify the same location at the same time and thus resulting in a much lighter editor design.

The idea of critical sections and locks is not novel but quite old actually. But this approach for collaborative editing has been debunked to be quite restrictive and counter intuitive to the "collaborative" aspect of collaborative text editing. This is where the novelty of our approach lies. We divide the document into various sections. If a client p wants to write to a section, she has to obtain a lock to a particular section and this is obtained once p is acknowledged by a majority of servers. If another node q wants to write to the same section, she has to wait for p to give up the lock or for p to be inactive long enough such that her lock is removed. This results in a lightweight collaborative editor LiteDoc whose shared documents consist of a series of single-writer multi-reader (SWMR) registers instead of one massive distributed list [8] . These registers and thus the documents created on LiteDocprovide deterministic guarantees of correctness, namely atomicity [9], [10].

C. Performance Study

Here, we present simulation results to back up our claim: There are fundamental engineering issues in the current approaches. We built a simple simulator to compare three approaches: single server, consensus-based, and SWMR register-based. Since we are focused on the performance of the underlying systems, we did not build a full-fledged editor. Instead, we use a workload generator to generate bunch of read and write events to the system, and compare the performance.

Simulation Setup: We perform our simulations on a single machine, equipped with 24 virtual CPUs and 48 GB memory on Google Cloud Platform (GCP) [11]. We achieve network virtualization through Mininet [12], a popular and realistic software to simulate datacenter network. We chose to use Mininet instead of deploying on physical machines because we want to understand the impact of network latency. Mininet allows us to add per-link (bidirectional from a host to the switch) artificial latency In Mininet, we set up a single switch topology, and each program runs on a virtual host that connects to the switch. The Mininet environment is quite stable, and the RTT (round trip time) between any two hosts is around 0.06 ms. Our Mininet setup script can be found on our Github repository [13].

For underlying database (that stores the content of the shared document), we chose Cassandra [14] and Etcd [15]. We use two versions of Cassandra, and they are the vanilla Cassandra (version 3.11) and Cassandra with the optimized ABD algorithm [16]. For Valina Cassan-

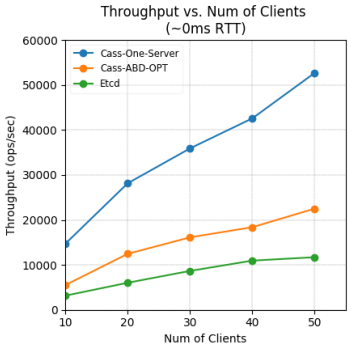

Fig. 1. Throughput

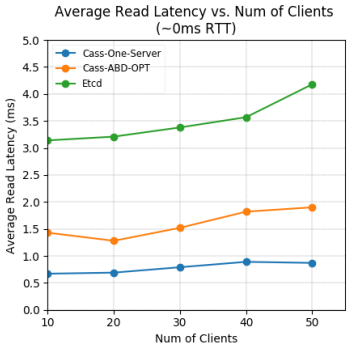

Fig. 2. Average Read Latency

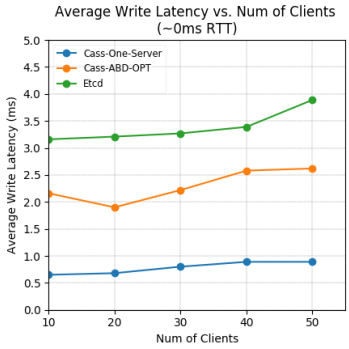

Fig. 3. Average Write Latency

dra, we set up a single server cluster to simulate the workload of single-server approach (Cass-One-Server), and for Cassandra equipped with ABD (Cass-ABD-OPT) algorithm, we set up a three-server cluster. Etcd internally uses Raft protocol [17] to achieve consensus, and we set up a three-server cluster as well. We chose Raft over Paxos [18], because the industry is gradually shifting Raft due to its easy-to-understand design and community support.

Workload Generator: The usual practice of benchmarking Cassandra involves using YCSB [19], but YCSB could produce concurrent writes and thus does not satisfy our semantic. We developed our benchmarking tool, programmed in Golang [20], which eliminates concurrent writes to the same location entirely. Our simulation program is available on our Github repository [21]. In each storage system, we use 50 key-value pairs (or SWMR registers) to model the shared document, where each key-value pair represents a section of the shared document.

Our workload simulator has two kinds of workloads, one varies the number of clients which performs database reads, and another varies the latency per link from a client host to the switch. In two workloads, each thread works on a single key-value pair. For the first workload, there are 10 client threads that perform reads and writes alternatively. The range of the read-only clients is $0, 10, 20, 30$, and 40. Each client thread performs no less than $10,000$ operations, and there is no artificial latency between a client and the switch (specified in Mininet). For the second workload, we vary latencies per client-to-switch links from 0 ms to 0.1 ms and 1 ms (by tuning on the Mininet script). Each simulation has 50 clients and lasts longer than 3 minutes to avoid system artifact (e.g., ramp-up and ramp-down effects). 10 out of 50 threads perform read-write operations, and the remaining 40 perform only read. The total write ratio in the system is 0.1. We have observed that the Mininet environment is quite stable; hence, our simulation are consistent across different runs of experiments (i.e., different parameter configuration).

We choose these parameters to reflect realistic scenarios. Particularly, if editor users are using the same close-by datacenter, then the client-to-switch link should have only minimal delay. Moreover, 5G is claimed to provide sub-ms latency between users and cell-towers. Hence, the 1ms configuration should capture the scenario of using 5G to access the shared document.

D. Performance

Figures 1, 2, and 3 present the results when there is no artificial latency. Not surprisingly, Cass-One-Server performs better than Cass-ABD-OPT and Etcd. In terms of throughput, Cass-One-Server is $2.3-2.6$ times higher than Cass-ABD-OPT and $4.5-4.7$ times higher than Etcd. Moreover, Cass-One-Server also has the lowest read and write latencies among all three setups. However, single-server approach has single point of failure. Cass-ABD-OPT has roughly 1.8x – 2x performance compared to Etcd-Raft. Moreover, one can see that Etcd-Raft is not that scalable. Performance decreases when there are 50 clients.

Tables I, II, and III present the results when there are artificial latencies introduced in Mininet. As expected, we see throughputs decrease when latencies increase as expected. Again, single-server approach has the best performance and Etcd-Raft has the worst performance, which align quite well with the theoretical expectations. An interesting observation is that even under relatively light workload (at most 50 clients), time to achieve consensus is expensive compared to communication time. For example, it takes roughly 1ms for clients to communicate with a server, but Etcd-Raft takes roughly 3.68 ms to reach an agreement on the order of events.

Our simulation setup actually favors Etcd. First, both vanilla Cassandra and Etcd are highly-optimized, as they have been used extensively in industry. Cassandra-ABD-OPT is not the case. Optimizing it is one of our future works. Second, we did not add latency between servers which allow Etcd to have the best performance. This is because round complexity of consensus algorithms are higher than SWMR register algorithms. The primitive results here motivate us to present a lighter and simpler design built on top of SWMR registers.

	0 ms	0.1 ms	1 ms
Cass-One-Server	52690.58	47880.01	29119.62
Cass-ABD-OPT	22509.36	21686.86	18239.17
Etcd	11707.41	11651.73	10438.1

TABLE I
THROUGHPUTS (OPS/SEC) VS RTT

	0 ms	0.1 ms	1 ms
Cass-One-Server	0.87	0.98	1.69
Cass-ABD-OPT	1.9	1.98	2.44
Etcd	4.18	4.21	4.68

TABLE II
AVERAGE READ LATENCY (MS) VS RTT

	0 ms	0.1 ms	1 ms
Cass-One-Server	0.89	1.01	1.7
Cass-ABD-OPT	2.62	2.71	3.05
Etcd	3.89	3.98	4.64

TABLE III
AVERAGE WRITE LATENCY (MS) VS RTT

III. LITEDOC FRAMEWORK

We begin with the architecture design of LiteDoc, followed by the correctness formulation and its benefits.

A. Architecture

Underlying Distributed System: We consider a set $\mathcal{S}$ of m servers that fully replicate the shared document. At all times, at most a minority, (i.e., $\leq \lceil \frac{m}{2} \rceil - 1$) of servers may fail by crashing. Every client and server in the system knows the value of m. Every document is split into n sections and each section is stored on an emulated single-writer multi-reader (SWMR) *atomic* register, $R_i : i = \{1 \cdots n\}$. Atomicity ensures that in an execution, every operation (read or write) seems to happen instantaneously within the invocation and response of the operation [9], [10]. Using a system terminology, an atomic register is a read/write data object (i.e., a key-value pair) that provides atomicity (or linearizability).

Every register R_i can be of arbitrary size ranging from one word to multiple pages. The choice of $n \in \mathbb{N}^+$ is specified during the creation of the document. Every client and server in the system knows the value of n. For now, we assume n is static and cannot be changed. In Section VI, we briefly discuss how to relax this constraint. In order to read the entire document, all m registers are read in a pre-defined order: $R_1, R_2, \cdots, R_n$. The granularity of the document can be determined by the client's needs. It is important to note that the granularity of the document will affect the performance of the system. A more thorough performance study is left as an interesting future work

Clients: A dynamic set of clients (or editor users) can be served by the servers. These clients may enter and leave the system at will. From a theoretical point of view, any number of clients can read from these registers, because they do not affect or modify the system state. In practice, this is limited by the performance of the underlying database. If it goes beyond some limit, then we can use simple load balancing/management service to address the issue.

At all times, at most one client (writer) is allowed to perform a write operation (or update) to any register. A write operation can involve any update to the document: insert, delete, paste, undo, font change, etc. This writer client for a particular section (register) can change from time to time as long as no more than one client writes to a particular section. Thus every register is considered a critical section and only one writer is allowed to enter it. Any client can fail by crashing at any time. If a client crashes, her changes may or may not be committed to the shared document.

Writer Lock: Recall that we avoid concurrent writes by allowing one writer for a single register in LiteDoc. If a client p wants to write to a section (register), it will first have to ask permission for a writer-lock to do so. Once p gets an acknowledgment for the writer-lock from at least $\lfloor \frac{m}{2} \rfloor + 1$ servers, it can start modifying the section. If a different client q is already writing to a section, p fails to obtain the writer-lock and has to either wait for q to release the lock or write to a different section of the document. Every lock comes with a pre-set timer for t time units which expires in case the current writer p goes idle. We need the timer not only for idle clients but also for clients that crash when they are inside the critical section, in order to prevent lockout. Client p will be notified before the timer expiry and asked if she wants to continue. A default value of t will be specified during document creation. It can also be requested by the writer during obtaining the lock.

B. Consistency Conditions and Correctness

In order to formalize the specification of a collaboratively edited document, Attiya et al. [8] introduced the specification of a *replicated list* object, which allows its users to insert and delete elements at different replicas. The authors provide a strong and a weak version of the same replicated list. The strong version ensures that orderings (attained by consensus) relative to deleted elements hold even after the deletion and the weak version offers no such guarantees. The replicated list is a simple data structure to reason with and offers eventual consistency to all its users.

We propose a different way to argue about correctness. Shared documents created using LiteDoc are divided into several SWMR atomic registers. As atomicity is a composable property, it is easy to see that the resulting documents support atomic reads and writes. This is also more natural for editor users, since atomicity

creates an illusion that concurrent users are operating the system on a single machine. In other words, we can build on top of existing theoretically proven SWMR algorithms, e.g., ABD algorithm [16], and writer-lock (a sub component of Paxos). The full proof will be presented in the future technical report.

C. Benefits

By simplifying the supported semantics, LiteDoc provides following benefits over existing editors:

- *Speed*: As we saw in Section II, SWMR registers have higher performance. Plus, we avoid the usage of computationally intensive algorithms like OT or diff-sync. From a theoretical sense, LiteDoc should have better performance.
- *Scalability*: As consensus is known to scale poorly, SWMR registers do not have such issue. We believe LiteDoc will be able to achieve close to linear scalability (i.e., similar to Cassandra, the core system that we are developing on).
- *Robustness*: As argued previously, we can formally specify and prove correctness as long as majority of the servers are fault-free.
- *Ease-of-Development*: There are two main difficulties in developing collaborative editors, consensus and consistency maintenance (e.g., OT or diff-sync). LiteDoc does not need those two primitives, because conflicts are removed at the source
- *Stronger guarantees*: It is important to note that LiteDoc guarantees a stronger consistency condition than eventual guarantees provided by most existing collaborative editors (more details to be discussed in Section V). This is because that all operations are seen by a quorum of servers in LiteDoc; hence, all updated will be always available as long as majority of servers are fault-free.

IV. UI/UX Design

We present our initial user interface (UI) design. A screenshot is presented in Figure 4. LiteDoc will provide comparable editing functionality to existing collaborative editors and will have the familiar horizontally spanning menu bar and editing tool bar. With the introduction of critical sections, each client will have a unique color and name associated with it. Each section will be outlined in the color of the client working on that section. A user should know with ease which section they are working in as denoted by the corresponding color. The color-coded bar, user photo, and dashed box are served as visual cues to remind users whether they have the editing (update) right at a particular location.

V. Related Work

Most existing collaborative text editors aim to guarantee eventual consistency, i.e., if users stop modifying the document, then the replicas will eventually converge to the same state. These systems use either a

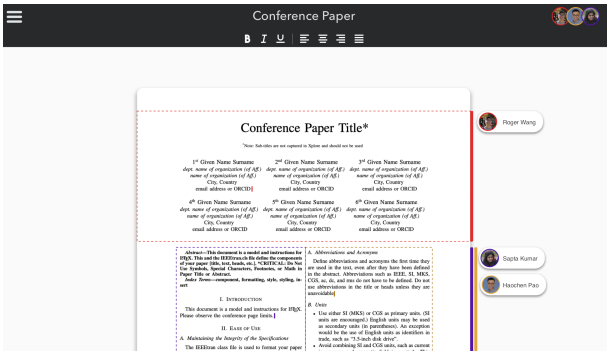

Fig. 4. LiteDoc UI Layout

single server to order these events or run a consensus algorithm among various servers to attain agreement on ordering of events. As a result these applications might not be available in the event of server failures. Theoretically speaking, the shared document can be considered to be *replicated list*. Attiya et al. [8] provided the specifications of the replicated list data structure (Strong and Weak), which models the core functionality of replicated systems for collaborative text editing. The authors showed that for a large class of list protocols, implementing the list specification requires a metadata overhead that is at least linear in the number of elements deleted from the list. The class of protocols to which this lower bound applies includes all list protocols mentioned above. Ignat et al. [22] provided a good evaluation and comparison of the centralized wikis system and several peer-to-peer collaborative text editors, based on several qualitative and quantitative metrics

Once an ordering on the update events is determined, the application updates the various local copies using complicated techniques like general operational transformation [1], [2], context-based operational transformation [23], admissibility-based operational transformation [24], differential synchronization [25] etc. These techniques apply the new updates from the server on top of the already existing user changes on the local copy of the document. CRDTs (Conflict-Free Replicated Data Types) introduced by Shapiro et.al [6] assume updates to be commutative thus making the problem of collaborative editing relatively easier to handle. Oster et al. [26] provided an algorithm to make most updates to a document commutative on a peer to peer network. The delete event does not delete a character. Instead, it strikes out the character to be deleted thus storing everything from the beginning with a strikeout for the elements deleted. Consistency is maintained if the following three properties [2] hold: intention preservation, causal consistency and convergence. Pacull et al. designed a system model called Duplex [27] which uses document decomposition. These authors also assume the existence of a kernel, shared by all members of the collaboration. The model is based on splitting the doc-

ument into independent parts, maintained individually by the owners and replicated by a kernel. They provide recovery mechanisms in case of failures or divergence from co-collaborators. Greif et al. [28] developed a collaborative editing system that builds on top of a language Argus, that provides support for a transactional storage system which inherently require consensus to complete. Minör and Magnusson [29] presented a model for semi-synchronous collaborative editing which fills the gap between asynchronous and synchronous editing styles. Their model is based on hierarchically partitioned documents, fine-grained version control, and active diffs for supplying collaboration awareness. Kaki et al [30] proposed the use of *invertible relational specifications* of an inductively-defined data type in order to understand important aspects of the data type such that different instances can be safely merged in a replicated environment. To the best of our knowledge, no one has taken our approach – simplifying semantics and using theoretically proven distributed primitives to build a fast, scalable, and robust editor.

VI. Conclusion and Future Work

We present the design of LiteDoc, and argue why it can achieve higher speed, scalability and theoretically proven robustness. We are implementing LiteDoc, and will opensource our work once it is fully tested.

We summarize several interesting future works below.

Benchmark Tool: We would like to build a tool to stress test collaborative editors. Particularly, we want to numerically analyze how well each editor behave under different latency, concurrency, and/or failures.

Dynamic Sections: The number of atomic SWMR registers and their sizes are customizable (can range from one word to multiple pages) and can be modified on the fly. In order to modify the section sizes and to add or remove sections, we have to run a consensus protocol on all the servers. Typically these will be a rare events when compared to the number of read/write operations on the document. An alternative is users can coordinate among themselves, and have a person manually configure the setup.

Peer-to-peer Systems: Most existing collaborative editors are built on top of server-client model (i.e., in the cloud environment), because they rely on expensive consensus algorithms, and consistency maintenance algorithm has poor performance with very high concurrency or dynamism, typical scenarios in peer-to-peer systems. Our approach is different. There are known theoretical algorithms for implementing SWMR registers in dynamic system. Therefore, we believe our design can be adapted to peer-to-peer systems.

References

[1] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, pp. 399–407, 1989.

[2] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 63–108, 1998.

[3] https://docs.google.com, "Google docs,"

[4] https://etherpad.org/, "Etherpad,"

[5] https://overleaf.com, "Overleaf,"

[6] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pp. 386–400, 2011.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[8] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski, "Specification and complexity of collaborative text editing," in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pp. 259–268, 2016.

[9] L. Lamport, "On interprocess communication. part I: basic formalism," *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986.

[10] L. Lamport, "On interprocess communication. part II: algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[11] https://cloud.google.com/, "Google cloud platform,"

[12] http://mininet.org/, "Mininet,"

[13] https://github.com/haochenpan/LiteDoc Mininet, "Litedoc-mininet,"

[14] http://cassandra.apache.org/, "Cassandra,"

[15] https://etcd.io/, "etcd,"

[16] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, pp. 124–142, Jan. 1995.

[17] https://raft.github.io/, "The raft consensus algorithm,"

[18] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998.

[19] https://github.com/brianfrankcooper/YCSB/, "Yahoo! cloud serving benchmark,"

[20] https://golang.org/, "The go programming language,"

[21] https://github.com/haochenpan/LiteDoc Workloads, "Litedoc-workloads,"

[22] C. Ignat, G. Oster, P. Molli, M. Cart, J. Ferrié, A. Kermarrec, P. Sutra, M. Shapiro, L. Benmouffok, J. Busca, and R. Guerraoui, "A comparison of optimistic approaches to collaborative editing of wiki pages," in *Proceedings of the 3rd International Conference on Collaborative Computing: Networking, Applications and Worksharing, White Plains, New York, USA, November 12-15, 2007*, pp. 474–483, 2007.

[23] D. Sun and C. Sun, "Context-based operational transformation in distributed collaborative editing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 10, pp. 1454–1470, 2009.

[24] D. Li and R. Li, "An admissibility-based operational transformation framework for collaborative editing systems," *Computer Supported Cooperative Work*, vol. 19, no. 1, pp. 1–43, 2010.

[25] N. Fraser, "Differential synchronization," in *Proceedings of the 2009 ACM Symposium on Document Engineering, Munich, Germany, September 16-18, 2009*, pp. 13–20, 2009.

[26] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for p2p collaborative editing," pp. 259–268, 2006.

[27] F. Pacull, A. Sandoz, and A. Schiper, "Duplex: A distributed collaborative editing environment in large scale," pp. 165–173, 1994.

[28] I. Greif, R. Seliger, and W. E. Weihl, "Atomic data abstractions in a distributed collaborative editing system," in *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pp. 160–172, 1986.

[29] S. Minör and B. Magnusson, "A model for semi-(a)synchronous collaborative editing," in *Third European Conference on Computer Supported Cooperative Work, ECSCW'93, Milano, Italy, September 13-17, 1993, Proceedings*, p. 227, 1993.

[30] G. Kaki, S. Priya, K. C. Sivaramakrishnan, and S. Jagannathan, "Mergeable replicated data types," *PACMPL*, vol. 3, no. OOPSLA, pp. 154:1–154:29, 2019.