Log-Structured Conflict-Free Replicated Data Types

UCSB Technical Report 2021-01

Nazmus Saquib, Chandra Krintz and Rich Wolski
Department of Computer Science
University of California, Santa Barbara
{nazmus, ckrintz, rich}@cs.ucsb.edu

***Abstract*—Conflict-Free Replicated Data Types (CRDTs) are highly available data types conforming to strong eventual consistency. CRDTs do not need synchronization for concurrent updates and can resolve conflicts locally without extra coordination. In this paper, we introduce Log-Structured Conflict-Free Replicated Data Types (LSCRDTs) that leverage the advantages of both CRDTs and append-only logs. We show that a log-based approach addresses several well-known challenges to using CRDTs. First, it facilitates a uniform approach to reversing operations. Second, it removes constraints such as exactly-once delivery and idempotence of operations which are commonly associated with operation-based CRDTs. Third, it provides robustness via efficient access to earlier versions of the data types. We select three commonly used CRDTs for our study: register, counter, and set. We show how each is implemented using LSCRDT, demonstrate the benefits achieved, and present an empirical evaluation of their performance.**

***Index Terms*—CRDT, data structure, replication, strong eventual consistency**

I. Introduction

Distributed systems often replicate data for low latency and high availability. The shared state among the replicas is maintained according to different system models. For example, strong consistency requires a replica to coordinate with other replicas to execute an operation. Coordination increases latency and an operation may fail as a result of a network partition, preventing the necessary communication between replicas. Weaker consistency models such as eventual consistency enable replicas to execute an operation locally and asynchronously propagate the operation to other replicas. This results in lower latency but with a temporary divergence in replica views that must be eventually reconciled.

Conflict-Free Replicated Data Types (CRDTs) [1]–[3] are abstract data types that provide a principled approach for this asynchronous reconciliation. CRDTs support a weaker model than strong consistency, namely, Strong Eventual Consistency (SEC) [1]. SEC guarantees that whenever two replicas receive the same set of updates, they reach the same state.

Broadly, there are two types of CRDTs: state-based and operation-based (or op-based) [1]. In state-based CRDTs, an operation is executed on the local replica state. A replica periodically propagates its state to other replicas to achieve consistency. A disadvantage of this approach is the communication overhead associated with shipping the full state, which at times can be large. In op-based CRDTs, an operation is executed on the local replica and the operation is asynchronously propagated to other replicas. Although operation-based CRDTs do not communicate state, they require exactly-once causal broadcast. Delta State Conflict-Free Replicated Data Types (δ-CRDTs) [4] combine the advantages of state-based and op-based CRDTs. Like the state-based, δ-CRDTs can tolerate unreliable networks and, in particular, do not require exactly-once causal broadcast as a communication network property. However, like the operations-based approach, they do not require that the full replica state be communicated, but rather, they communicate only state changes or "deltas".

In this work, we investigate Log-Structured Conflict-Free Replicated Data Types (LSCRDTs), a new form of data type aimed at combining the advantages of append-only logs with CRDTs. LSCRDTs implement SEC, and like op-based CRDTs, communicate only operations rather than replica state or replica state deltas. However, like state-based and δ-based CRDTs, LSCRDT does not rely on strong network reliability properties (such as exactly-once causal broadcast).

Our choice of using logs to facilitate this functionality is driven by the multiple benefits that logs inherently provide. First, logs facilitate a uniform approach to reversing operations, which, to the best of our knowledge, is an open problem for CRDTs [2]. As the execution of an operation can be represented by appending entries to one or more logs, reversing the operation is equivalent to trimming one or more logs. This in turn enables us to implement arbitrary non-commutative data types: if two operations do not commute we can rollback log (i.e. reverse/undo an operation) and execute them in the desired order. Therefore, we are not bound by the strict requirement of using a commutative data type or using a variant of a data type that is commutative (e.g. two-phase variant of set) as in the case of CRDTs.

Second, although LSCRDT is op-based it is not bound by the restrictions generally associated with op-based CRDTs such as exactly-once causal delivery or operation idempotence. In op-based CRDTs, when a replica executes an operation it asynchronously sends a message denoting this execution to all other replicas. In our approach, each replica logs the executed operation along with a unique *version stamp* in causal order. Each replica periodically reads from other replicas' logs following this log order (e.g. in the case of Kafka [5] logs each replica reads monotonically increasing *offsets*). Because log order reflects the causal order, LSCRDT maintains causal order by ensuring log order is maintained during reads. Moreover, each operation is version stamped uniquely so duplicate operations can be easily identified and

ignored. As a result, our implementation of LSCRDT can be deployed over unreliable networks.

Third, logs are append-only data structures – old data is always retained and new data is appended to the log. As such, updates are inherently coordination free and past *versions* of the data structure are programmatically accessible. Whenever an *update* operation is executed on a data type (e.g. increment/decrement on a counter), a new version is created. Versioned data types are useful for a variety of distributed systems challenges including data repair [6].

While the choice of using logs provides us with the above features, the choice of using op-based replication provides us another added feature: *generalization*. State-based replication generally requires *type-specific* join algorithms. On the other hand, our approach relies on forming a consistent order of operations, irrespective of the underlying data type. As we will see in Section IV, agreeing upon a consistent order of operation is not type-specific in LSCRDT. Thus, the LSCRDT approach is more extensible than state-based approaches.

In this paper, we describe LSCRDT and an example implementation that uses an open-source runtime system that supports a distributed log abstraction. We use this implementation to compare the latency and throughput for multiple, mixed operation, workloads for popular CRDT building blocks (e.g., counter, register, set) between LSCRDT and δ-CRDT. We find that LSCRDT introduces latency overhead by less than 1.6x in the worst case over δ-CRDTs while providing the additional advantages associated with a log-based approach described previously. Further, the throughput overhead introduced by LSCRDT is less than 1.3x in the worst case over δ-CRDTs. Our results also show that LSCRDTs are scalable, that accessing prior versions of data types such as counters and registers is just as fast as accessing their latest versions, and that replicas converge faster when updates are periodic and infrequent.

II. System Model and Overview

We consider a distributed system of N replicas. Each replica is assigned a node ID from a set S. We represent a replica as $X_s, s \in S$. The underlying network is asynchronous and unreliable; messages may be dropped, duplicated, or reordered. The network may partition and eventually recover. Each replica has local durable storage. We assume replicas may face non-byzantine failures; a replica may crash but will have access to the information recorded in durable storage up to the point of failure once it recovers.

Over time replicas may diverge from each other due to update requests from clients that are processes that can mutate or query a data type by sending requests to any replica. To reconcile this divergence each replica periodically performs a round of *merge steps* with the other replicas. A merge step is always between a pair of replicas. Therefore, in a round, there are at most $N - 1$ merge steps. In a merge step, one replica (known as the *reader*) reads entries in the log of operation from another replica (known as the *source*). The goal of the merge step is for the reader to identify and incorporate operations unknown to it that the source has already executed. The reader ensures that the causality relationship among the operations is retained while creating this merged list of operations and subsequently executing them. We present the details of the merge step in Section IV. Note that in merge steps readers at times may have to rollback some operations and re-execute them along with new operations. As long as the replicas execute operations in the same log order, replicas will converge irrespective of operation commutativity, similar to the replicated state machine concept used in consensus algorithms such as Raft [7].

Although our approach requires log rollback to maintain the order of operations, it does not require any locking mechanism among replicas to agree upon a unique order of operation execution. This order can be found just by observing the timestamped operations of the source as described in Section IV. Any partial order formed during merge steps maintains the causal order observed within the total order. Rollbacks provide two capabilities to LSCRDTs: (i) implementing arbitrary non-commutative data types and (ii) maintaining a global version history consistent among all replicas. Note that as an optimization (not explored in this paper) if the underlying data type is commutative and a global version history is not needed LSCRDT can store state deltas (like δ-CRDTs) and, thus, remove rollbacks (and their performance impact) altogether. We plan to investigate this optimization as part of our future work.

Various algorithms have been proposed to maintain order in list or sequence CRDTs such as Logoot [8], LSEQ [9], RGA [10], Treedoc [11], and WOOT [12]. Our method to maintain order among logs of operations is an adaptation of [13], which is based on RGA [10]. This approach provides us a uniform way to create a replicated data type irrespective of the operations supported by it; once we establish a common order of operations among all the replicas our system will converge. Although RGA-based approach has been used in other data types such as JSON [13], the use of logs introduces a new performance challenge – avoiding log scans. We explain how we can avoid full log scans in Section IV.

III. Data Types Using Logs

In this section, we explain how LSCRDTs are stored on logs. We present the replication process in Section IV. We choose three data types widely studied in CRDT literature for this exposition: registers, counters, and sets [14]–[18]. Our approach is agnostic of the underlying log storage system. We assume entries in a log can be addressed by monotonically increasing *sequence numbers* (e.g. *offset*s in the case of Kafka [5], *LSN*s in the case of Facebook LogDevice [19], and *sequence number*s in the case of CSPOT [20]). We further assume the log storage system exposes functionalities (i) to create logs with a given name, (ii) to write to a specified log and get the sequence number corresponding to the write on success, (iii) to read from a specified log at a given sequence number, (iv) to retrieve the latest sequence number of a specified log, and (v) to trim the log up to a specified

sequence number i.e. all entries with greater sequence numbers are removed. As long as these criteria are met, we can use any log storage system.

LSCRDTs tag each operation performed on a data type with a *version stamp* (Lamport timestamp), which is a concatenation of a counter and a node ID drawn from S. We represent the counter and node ID of a version stamp vs as $vs.counter$ and $vs.nodeID$, respectively. We say version stamp vs_a is less than version stamp vs_b ($vs_a < vs_b$) if (i) the counter of vs_a is less than that of vs_b, or (ii) both the counters are same but node ID of vs_a is less than that of vs_b. When replica X_s executes a new operation in response to a client request, it tags it with version stamp vs ($vs.nodeID = s$), which is greater than all other version stamps it has observed so far (operations that *happened before*). Thus if operation op_a happens before op_b, $vs_a < vs_b$ where vs_a and vs_b are the version stamps of operations op_a and op_b, respectively.

Version stamps of concurrent operations can be ordered arbitrarily but deterministically. Throughout the rest of the paper, we use version stamps to refer both to the version stamp itself and to the operation it tags. The intended use will be clear from the context. We say vs is an operation of X_s (alternatively, X_s is the originator of vs) if $vs.nodeID = s$.

A. Register

The register data type maintains a single value (e.g. an integer, an object, etc.). It supports two operations, *assign* to set a value and *retrieve* to get a value. We introduce *OpLog* to store all the update operations, in this case, assigns. There is one OpLog per replica. We represent the OpLog of the replica X_s as $OpLog(X_s)$. As all the update operations are recorded in the OpLog, a data type can be reconstructed up to a certain version if required. Replicas can read each other's OpLogs to create a merged list of all the update operations. As the retrieve operations do not update the register, we do not have to record those. Each entry in an OpLog is the tuple *(vs,op,val)*. *vs* is the version stamp of the operation, *op* is the type of update operation (in case of register there is only one, i.e., assign), and *val* is the operand of *op*.

To execute an assign operation a replica writes the appropriate entry to the OpLog. For example, suppose a request comes to X_A to assign the value 5 to a register. We further assume the greatest counter value among all the version stamps X_A has seen so far is 2. Then X_A writes the entry $(3A, assign, 5)$ to its OpLog to execute the operation. To respond to a retrieve request, it simply reads the last entry of the OpLog and returns the *val* field of that entry. If some previous version stamp vs_i is supplied as an argument of value, the replica can search the OpLog to locate the entry with the *vs* field equal to vs_i.

To make this search efficient, LSCRDT maintains an in-memory map from version stamps to sequence numbers in OpLog. This approach has been used in other log-based systems as well, such as Riak Bitcask [21]. Note that this is an optimization and not necessary for the correctness of the system. This in-memory map is populated at startup and can be reconstructed at any time. Any update to the register is first appended to the log and then a map entry is created.

B. Counter

The counter data type supports increment (*inc*), decrement (*dec*), and *retrieve* operations. Like register, a counter also has an OpLog. However, it maintains one additional field per entry for the cumulative sum to avoid log scans while computing the counter value corresponding to a version. For example, assuming the initial value of a counter is 0, if replica A first increments the counter by 5, next decrements the counter by 2, and finally increments it by 1, the entries of the OpLog will be $(1A, inc, 5, 5)$, $(2A, dec, 2, 3)$, and $(3A, inc, 1, 4)$. To find the latest value of the counter, the replica can now return the value in the last field of the last entry in the OpLog. Similar to register, an in-memory map can expedite the response to a retrieve request corresponding to an earlier version.

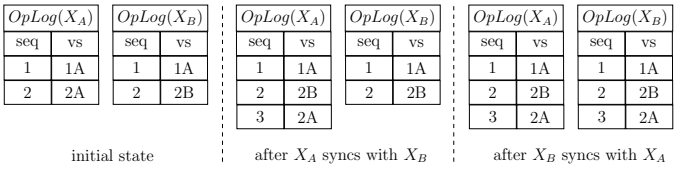
C. Set

Our set data type supports *add* and *remove* update operations and *in* and *all* read operations. Note that although CRDTs resort to using some variant of sets such as two-phase set (2P-set), grow-only set (G-set) etc; LSCRDT set works like a conventional set due to its capability to find a consistent ordering of non-commutative operations. The structure of OpLog of set is similar to that of register. However, set is different from register and counter in that each version is a collection of elements rather than a single value. Note that to reconstruct a set upto the latest version, we must scan the OpLog from the top. To avoid such expensive operations, we cache the elements in the set after every $cp\_interval$ number of operations using a second log. To make the search for the latest version fast, we also keep a copy of the latest operations that have not yet been checkpointed in memory. This in-memory list of operations is purged everytime we checkpoint our progress. Therefore, we have at most $cp\_interval - 1$ operations cached in memory at a time (which users can set). Setting the $cp\_interval$ to a low value makes queries faster but uses more space. Doing so may also decrease write throughput due to more frequent checkpointing.

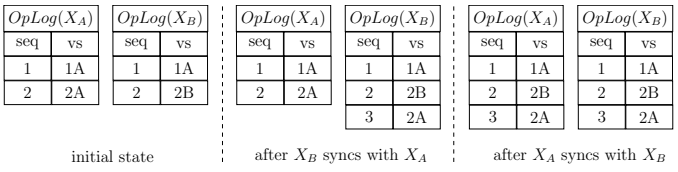
Note that to query a previous version of the set, we might need to access OpLog. For example, suppose $cp\_interval$ is set to 100, we have already executed 400 operations and we want to query the 257th version of the set. In this case, we first need to read the entry that was checkpointed after the 200th operation and then read the 57 following operations from the OpLog to reconstruct the desired version of the set. As we will see later in Section IV, a version of a data type might change due to updates from other replicas. In that case, the checkpoint that contains that version and the following checkpoints must be overwritten.

IV. Merge Step

Many data types have operations that do not commute (e.g. add and remove of the same element in a set). To achieve a consistent state for *replicated* data types, we must impose a



(a) X_A merges with X_B then X_B merges with X_A.



(b) X_B merges with X_A then X_A merges with X_B.

Fig. 1: Change in OpLogs as replicas merge with each other. We notice the two different sequence of merge steps results in the same consistent state at the end.

total order on the execution of operations [22]. An alternative to imposing a total order for arbitrary non-commutative data types is to switch between stronger and weaker form of consistency [23]–[26]. However, as discussed in Section II, a total order also helps us maintain a consistent version history among all replicas. In our work, we model the history of operations (OpLog) as a list CRDT and use an adaptation of the method used in [13] which is based on RGA [10] for maintaining order in the list i.e. the OpLog.

When a replica X_A executes an operation as a direct request from a client, it appends the operation at the end of $OpLog(X_A)$. Apart from direct client requests, replicas also execute operations that are unknown to them from other replicas' OpLogs. Assume vs_{new} is an operation in $OpLog(X_B)$ that X_A has not yet executed. We denote the operation immediately preceding vs_{new} in $OpLog(X_B)$ as vs_{pred}. As the intention is to maintain a consistent order of operations, X_A tries to place vs_{new} in its own OpLog after vs_{pred} as well. Therefore, to incorporate the unknown operation vs_{new}, X_A first locates vs_{pred} in $OpLog(X_A)$. Let us denote the operation in $OpLog(X_A)$ immediately succeeding vs_{pred} as vs_{succ}. That is, vs_{new} and vs_{succ} are concurrent operations. Now X_A inserts vs_{new} in $OpLog(X_A)$ immediately after vs_{pred} if $vs_{new} > vs_{succ}$. Otherwise, X_A skips over all contiguous version stamps that are greater than vs_{new} and then places vs_{new}. Of course, it might happen that vs_{pred} is not present in X_A to begin with. In that case vs_{pred} must be inserted first. This implies that X_A should start reading $OpLog(X_B)$ from the earliest sequence number that contains an operation unknown to it. We express this whole procedure of inserting operation vs_{new} after vs_{pred} as $insert(vs_{new}, vs_{pred})$.

To illustrate how *insert* works, we refer to the OpLogs in Figure 1 (only the sequence numbers and version stamps are shown for brevity). We consider two replicas in our system, X_A and X_B. Let us assume X_A executed operation $1A$ that X_B became aware of during the latter's merge step. At this point, both X_A and X_B executed one operation independently but concurrently, operation $2A$ and $2B$ respectively. Now we consider two different scenarios. (i) Figure 1a. X_A (reader) merges with X_B (source). For now, we assume readers start comparing the two OpLogs from the beginning (we show in Section IV-B how full log scans can be avoided). Both OpLogs have $1A$ as the first entry, so no action is needed. However, X_B has $2B$ in the second entry whereas X_A has $2A$. This is equivalent to the insert operation $insert(2B, 1A)$ i.e. insert $2B$ after $1A$ in $OpLog(X_A)$ (as $2B$ comes after $1A$ in $OpLog(X_B)$). We note how the insertion operation is implicitly embedded in the log order. As X_A currently has $2A$ after $1A$ and $2B > 2A$, it can place $2B$ after $1A$. "Placing $2B$ after $1A$" is a multi-step process: X_A trims its OpLog up to sequence number 1, append the entry containing $2B$, and finally re-append the entry containing $1A$. Additionally, it trims/(re-)appends to any logs used by the underlying data type. When X_B (reader) merges with X_A (source) after this, X_B can simply append $2A$ after $2B$ in its OpLog. (ii) Figure 1b. X_B (reader) merges with X_A (source). Starting comparison from the top of the OpLogs as before reveals different entries in the second entry: $OpLog(X_A)$ has $2A$ as the second entry whereas $OpLog(X_B)$ has $2B$. This translates to the operation $insert(2A, 1A)$ to be executed in OpLog of X_B. As the version stamp after $1A$ at X_B is $2B$ and $1A < 2B$, $1A$ is placed after $2B$. Merging the other way follows the steps similar to the previous scenario. We see that in both scenarios we end up with the same final state in both the replicas.

To understand how this replication method converges, we note a few points. First, the order of an operation in an OpLog either remains unchanged or is pushed down, but never pulled up. This is due to how $insert$ works: it either appends a new operation at the end, in which case there is no change in the order of operations; or it inserts an operation in between existing operations, effectively pushing all the operations that follow down by one. This also implies that the relative order of two operations in a log once set remains the same. Second, $insert$ breaks tie between two concurrent operations arbitrarily but deterministically (the two concurrent operations are the new operation from the source and the current successor of the intended predecessor in the reader).

To see this, let us consider two concurrent operations vs_a and vs_b. Without the loss of generality, let us assume $vs_a > vs_b$. Now if vs_a has already been executed (i.e. vs_a is the successor of the intended predecessor in reader), vs_b skips over vs_a. Therefore vs_a comes before vs_b. On the other hand, if vs_b has already been executed (i.e. vs_b is the successor of the intended predecessor in reader), vs_a can be placed in its place, effectively pushing down vs_b. Therefore, vs_a comes before vs_b in this case as well. Finally, a reader always reads the operations unknown to it from the source in monotonically increasing sequence numbers. This means even when a reader has not observed all the operations executed by the different replicas in the system, its OpLog contains a partial order of the total order formed by all the operations (due to the first and the second points). Hence once the replicas observe all the operations, the system achieves consistency.

In a merge step between a reader X_i and a source X_j, the

reader performs two tasks: (i) *Conflict detection*: The reader detects whether it is in conflict with the source, i.e. whether the source has operations that the reader does not know of. Note that we are concerned with unidirectional conflict, i.e. if the reader has operations that are unknown to the source no extra steps are taken (this is resolved during some other merge step when the current source becomes a reader). (ii) *Conflict resolution*: In case of conflict, the reader resolves this conflict, possibly by reordering the operations which require rollback and replay of some operations. The conflict detection stage finds the sequence numbers of the two OpLogs from where the comparison should be started ($reader_{start}$ and $source_{start}$ for OpLog of the reader and the source respectively) to guarantee that the reader encounters all the operations it has not seen that have been already executed by the source. These two sequence numbers are used by the conflict resolution stage to incorporate all the unknown operations in the reader's OpLog and thus the underlying data type.

We next introduce a new log that helps us to avoid full log scan (Section IV-A). We show how the conflict detection stage uses this log to detect the presence and point of conflict (Section IV-B). Section IV-C then describes how the conflict resolution stage takes this point of conflict information and uses $insert$ operations to execute a list of ordered operations.

A. KnowledgeLogs

OpLogs grow over time and the merge steps become costly if we must scan from the top. To avoid a full scan of the OpLog of the source by the reader, a replica maintains a map of the last observed version stamp from each replica to a sequence number in its OpLog using one KnowledgeLog for each replica. Each entry in a KnowledgeLog contains the tuple *(vs, op_seq)*. *vs* denotes the version stamp of the operation. *op_seq* denotes the sequence number of OpLog where the operation with *vs* was first appended. More precisely, each entry of KnowledgeLog K_i^j on host X_i contains tuples that map each version stamp vs whose node ID is j to a sequence number in $OpLog(X_i)$. Although the position of a version stamp might change due to later merge steps, note that a version stamp can only be pushed down in order but never pulled up due to the way $insert$ works. Therefore, the sequence numbers stored in KnowledgeLogs provide us a starting point to search for a version stamp. The version stamp might be at that sequence number, or at a later one, but never at an earlier one.

We refer to Figure 2 as an example of interactions among OpLogs and KnowledgeLogs. Operation $1A$ is inserted in $OpLog(X_A)$ at sequence number 1. To record the mapping from version stamp $1A$ to sequence number 1, X_A appends $(1A, 1)$ to K_A^A. Similarly, X_A appends $(2A, 2)$ to K_A^A to record that the operation with version stamp $2A$ was inserted in $OpLog(X_A)$ at sequence number 2. A merge step with X_B results in the operation with version stamp $2A$ to be pushed down in order i.e., at sequence number 3. As we have already recorded $2A$ in K_A^A and we can reach $2A$ in $OpLog(X_A)$ even if we start scanning from the recorded *op_seq* value (in this case 2), we can keep it unchanged. We only append the entry $(2B, 2)$ to K_A^B. Now if X_A (reader) performs a merge step with an arbitrary replica X_s (source) and wants to know whether X_s has any operation originating from replica X_B that the reader does not know of, it can simply compare the tails of K_A^B and K_s^B. If the last entry of K_A^B contains a version stamp that is less than that of the version stamp contained in the last entry of K_s^B, then X_s has operation originating from X_B that X_A does not know of (as two version stamps with same node ID follow happens-before relationship and version stamps are written to the KnowledgeLog in increasing order). This process is explained in detail in the next section.

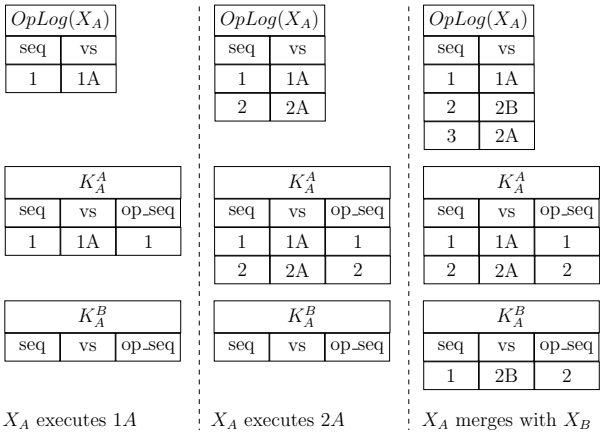


Fig. 2: Mapping from version stamps to sequence number of OpLog in KnowledgeLogs.

B. Conflict Detection

In the conflict detection stage during a merge step between reader X_i and source X_j, the reader X_i compares the last entries of K_i^m and K_j^m, $\forall m \in S$. We represent the last entry of a log L by $tail(L)$ and a field f in entry e by $e.f$. If $tail(K_i^m).vs < tail(K_j^m).vs$, this means X_j (source) has executed operations that X_i has not. This holds as the operations in a KnowledgeLog have the same node ID and are executed in increasing order of their counter. The counter captures the happens-before relationship between two version stamps with the same node ID. We say X_i lags behind X_j with respect to X_m when $tail(K_i^m).vs < tail(K_j^m).vs$. X_i might lag behind X_j with respect to more than one replica. Let us represent the set of all replicas with respect to which X_i lags behind X_j as X_{lag}.

We represent the set of node IDs of the replicas in X_{lag} as S_{lag}. We find the replica X_p in X_{lag} such that $tail(K_j^p).op\_seq < tail(K_j^l).op\_seq, \forall l \in S_{lag} \wedge l \neq p$. That is, X_p is the replica whose operation is at the earliest point of conflict between X_i and X_j. However, X_i might not know about operations of X_p that have version stamps less than $tail(K_j^p).vs$. To ensure X_i can detect all unknown operations, it scans backward from the tail of K_j^p until it finds the entry e such that the entry before it has a version stamp equal to $tail(K_i^p).vs$. Then $e.op\_seq$ is the sequence number from which the reader start scanning the source's OpLog (i.e.

$source_{start} = e.op\_seq$). In other words, $e.vs$ is the earliest operation in $OpLog(X_B)$ that X_A has not yet executed.

We can prove that there can be no operation that is unknown to X_i in $OpLog(X_j)$ before $source_{start}$ by contradiction. Let us assume there is indeed a version stamp vs in $OpLog(X_j)$ at sequence number $source^{'}_{start}$ such that $source^{'}_{start} < source_{start}$ and $vs.nodeID = q$. That would mean one of the following: (i) $p \neq q$. That is, the tail of K^q_j contains an entry with op_seq value that is smaller than all other op_seq values of knowledge logs with respect to which X_i lags X_j. However, this is not possible, as we are taking the minimum of op_seq values of the tails of the relevant knowledge logs to find p. (ii) $p = q$. In that case, there must be an entry in K^p_j with vs greater than $tail(K^p_i).vs$ and op_seq less than $source_{start}$. However, this is not possible either, as we scan back to make sure we find the earliest version stamp in K^p_j that X_i has not seen. Hence we arrive at a contradiction and $source_{start}$ must be the earliest sequence number in $OpLog(X_j)$ where there might be an operation that X_i has not executed yet.

Assume the version stamp of the entry at sequence number $(source_{start} - 1)$ in $OpLog(X_j)$ is vs_{prev}. To incorporate $e.vs$, X_i executes $insert(e.vs, vs_{prev})$ in $OpLog(X_i)$. To do this, X_i first finds the sequence number of $e.vs$ in $OpLog(X_i)$ – the value of $reader_{start}$ is this sequence number plus one. Note that all operations in $OpLog(X_j)$ from sequence number 1 to $source_{start} - 1$ must be present in $OpLog(X_i)$, otherwise there is some operation between these two sequence numbers in $OpLog(X_B)$ that X_A has not seen, and the value of $source_{start}$ found by the previous steps would have been different. Therefore, $reader_{start}$ must be greater than or equal to $source_{start}$. To find the value of $reader_{start}$, X_i starts scanning $OpLog(X_A)$ from the sequence number $source_{start} - 1$. It stops scanning if the currently scanned entry has a version stamp equal to vs_{prev}. The required value of $reader_{start}$ is the sequence number where we stop scanning plus one.

To illustrate the conflict detection stage, we consider the scenario in Figure 3. Let us assume there are three replicas in our system, X_A, X_B, and X_C. The OpLog of X_A has $1A$ and $2A$, whereas the OpLog of X_B has $1A$, $2B$, $3B$, $4C$, and $2A$. One possible sequence of actions that might lead to this state: X_A executed operation $1A$. X_B merged with X_A, and then executed two operations $2B$ and $3B$. X_C (not shown in the figure) merged with X_B and executed $4C$. X_B merged with X_C. X_A executed operation $2A$. Finally, X_B merged with X_A again. Now let us consider X_A performs a merge step with X_B. Comparing the tails of K^m_A and K^m_B, $m \in \{A, B, C\}$, we see that X_A lags behind X_B with respect to X_B and X_C, i.e., $X_{lag} = \{X_B, X_C\}$ (we assume the absence of entry in a KnowledgeLog to be equivalent to having a placeholder entry with a version stamp with minimum possible invalid counter value, in this case, 0). As the op_seq value of $tail(K^B_B)$ (i.e. 3) is smaller than that of $tail(K^C_B)$ (i.e. 5), $X_p = X_B$. However, X_A is not yet certain $tail(K^B_B).vs$ is the earliest unknown version stamp. X_A scans K^B_B backwards to find the earliest unknown version stamp, which in this

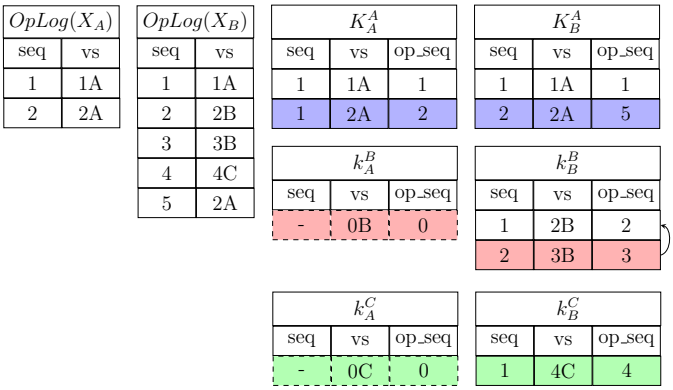


Fig. 3: OpLogs and KLogs of replicas X_A and X_B in a system with three replicas. Dashed entries represent placeholder entries used during computation when a knowledge log is empty. During conflict detection, the reader X_A compares the same colored entries with each other to find the earliest possible point of conflict. The arrow from the second entry to the first entry of K^B_B, represents X_A's backward scan to find the earliest version stamp with node ID B that it does not know of.

case is $2B$. The corresponding op_seq value is 2, therefore $source_{start} = 2$. The entry immediately preceding $2B$ in $OpLog(X_B)$ has the version stamp $1A$. X_A reads the entry at sequence number $source\_start - 1 = 1$ in $OpLog(X_A)$ and finds that the entry contains $1A$. Therefore $reader\_start$ is equal to $1 + 1 = 2$ as well. The conflict detection algorithm is presented in Algorithm 1.

C. Conflict Resolution

Conflict resolution is triggered when a conflict is detected, to find and execute a merged order of operations between the reader and the source. When there are one or more conflicts between the reader and the source, it rolls back the OpLog of the reader to the earliest point where the reader does not lag behind the source with respect to the version stamps before it and then replays the operations at the reader (adjusting the OpLog of the reader) to reflect the merged order. At the start of conflict resolution, X_i knows both $source_{start}$ and $reader_{start}$, i.e., the sequence number of $OpLog(X_i)$ and the sequence number of $OpLog(X_j)$ at which X_i should start comparing the two OpLogs. X_i creates an ordered list, R_i, of the operations in $OrdLog(X_i)$ starting from the sequence number $reader_{start}$ up to its latest sequence number. X_i creates a second ordered list, R_j, of the operations in $OrdLog(X_j)$ starting from the sequence number $source_{start}$ up to its latest sequence number.

To incorporate the operations unknown to itself, X_i first includes those operations from R_j to R_i by invoking $insert$ procedures: for each entry e in R_j, X_i first finds the entry e_{pred} in R_i which contains the version stamp immediately preceding e in R_j. If the version stamp of the entry following e_{pred} in R_i is smaller than $e.vs$, X_i inserts e immediately after e_{pred} (provided e is not already present there). Otherwise, it skips over all contiguous entries where the version stamp is greater than $e.vs$, and then inserts e (provided that e is not

Algorithm 1 Conflict Detection

Require: reader replica X_i, source replica X_j, and set of node IDs S
Ensure: earliest point of conflicts $source_{start}$ and $reader_{start}$

$X_{lag} \leftarrow \phi$
for $m \in S$ **do**
  if $tail(K_i^m).vs < tail(K_j^m).vs$ **then**
    $X_{lag} \leftarrow X_{lag} \cup \{X_m\}$
  end if
end for
if $X_{lag} = \phi$ **then**
  return
end if
$S_{lag} \leftarrow \phi$
for $X_m \in X_{lag}$ **do**
  $S_{lag} \leftarrow S_{lag} \cup \{m\}$
end for
$p \leftarrow \underset{m}{\operatorname{argmin}}(tail(X_j^m).op\_seq), m \in S_{lag}$
$idx \leftarrow latest\_seq(X_j^p)$
while $idx > 0$ **do**
  if $tail(K_i^p).vs < K_j^p[idx].vs$ **then**
    $source_{start} \leftarrow K_j^p[idx].op\_seq$
    $idx \leftarrow idx - 1$
  else
    break
  end if
end while
$vs_{prev} \leftarrow OpLog(X_j)[source_{start} - 1].vs$
$idx \leftarrow source_{start} - 1$
while $idx \leq latest\_seq(OpLog(X_i))$ **do**
  if $OpLog(X_i)[idx].vs = vs_{prev}$ **then**
    $reader_{start} \leftarrow idx + 1$
    break
  else
    $idx \leftarrow idx + 1$
  end if
end while
RESOLVECONFLICT($reader_{start}, source_{start}$)

Algorithm 2 Conflict Resolution

Require: reader replica X_i, source replica X_j, $source_{start}$ and $reader_{start}$ value obtained from Conflict Detection stage
Ensure: X_i is not lagging behind X_j

procedure RESOLVECONFLICT($source_{start}, reader_{start}$)
  $R_i \leftarrow \{OpLog(X_i)[reader_{start}], \ldots, tail(OpLog(X_i))\}$
  $R_j \leftarrow \{OpLog(X_j)[source_{start}], \ldots, tail(OpLog(X_j))\}$
  for all $e \in R_j$ **do**
    $e_{pred} \leftarrow$ the entry before e in R_j ▷ fixed dummy value assumed for first element
    $insert(e.vs, e_{pred}.vs)$ in R_i
  end for
  Prune $OpLog(X_i)$ starting from sequence number $reader_{start}$
  for all $e \in R_i$ **do**
    Replay/Execute $e.op$ and append e to $OpLog(X_i)$
    $q \leftarrow$ sequence number of e in $OpLog(X_i)$
    $k \leftarrow e.vs.nodeID$
    if $e.vs > tail(K_i^k).vs$ **then**
      append $(e.vs, q)$ to K_i^k
    end if
  end for
end procedure

already present there). Once X_i has all the operations in R_i, it rolls back, i.e., prunes, $OpLog(X_i)$ starting from $reader_{start}$ and then replays all operations in R_i at $OpLog(X_i)$. The conflict resolution algorithm is presented in Algorithm 2.

V. Handling Bounded Log Sizes

Logically, logs are append-only storages to which we can continuously append. Practically, we are bounded by the physical storage of our devices. Therefore, we can not record an unbounded number of versions of our data types. In this section, we describe how we can safely retain at least the last K versions of our data type by removing old entries from OpLog. We make two underlying assumptions: (i) our physical storage has the capacity to store more than K versions and (ii) the replicas merge among themselves at a rate such that there is at least one version common at the top of the OpLogs among all replicas before any replica runs out of space.

The major challenge in keeping a history of at least the last K versions is that this set of last K versions is constantly changing. The order of operations in the global history changes with updates from different replicas. So instead, we identify versions that we know for certain are not in the set of last K versions.

Consider an arbitrary version vs. Referring back to how our $insert$ operation of Section IV works, we know vs in the OpLog of a reader can be pushed down in order due to a merge step involving an unknown operation that the reader has not seen before. This essentially means that even if vs was not in the set of last K versions, it may become so. However, if all replicas in our system have observed all the same operations from the start up to vs, we know the positions of those versions will not change.

This means that a replica X_i can safely remove n number of operations from the top of its OpLog while preserving at least the last K versions if: (i) all the other replicas in the system have those n operations in the top n entries of their respective OpLogs and (ii) the replica has already seen at least K more new operations after those n operations. Note that X_i need not check whether the other replicas have received K more new operations to trim its own OpLog. It just has to make sure there is a set of operations from the top that all the replicas have executed in the same order. This trimming operation can be triggered after a certain number ($> K$) of versions have been recorded (which users can define). However, if this number is near K, trimming will be frequent and may adversely affect system performance.

A replica X_i need not scan the top of each replica's OpLog to find the set of entries that can be trimmed. Instead, it consults the tails of KnowledgeLogs to find such a set. For each node ID $s \in S$, X_i reads the tails of the N KnowledgeLogs $K_u^s (u \in S)$ and identifies the smallest version stamp. We denote this set of N versions as C. Then C contains one version vs for each originator X_s such that vs is the greatest version with node ID s that all the replicas have seen.

Next, X_i locates the earliest sequence number in its log

where such a version is present. Let this sequence number be seq. Note that any version in an entry preceding the entry at seq in $OpLog(X_i)$ must be present in all the other replicas. Therefore, X_i can trim all the entries up to seq provided that there are at least K versions following it. If not, it can backtrack the required number of entries to meet this condition and then proceed with trimming.

There are some caveats to this approach that we must overcome. First, although the earliest n versions trimmed in this approach are not part of the latest K versions, we may still need the last version of the earliest n versions as an anchor point for future $insert$ operations. This can happen if there is at least one replica such that it had exactly n versions at the time X_i trimmed its OpLog. Hence, we must record the last version in the set of trimmed versions every time we perform this operation. We can safely overwrite this record every time we trim the log.

Second, the mappings of version stamps to sequence numbers of OpLogs stored in KnowledgeLogs will be off by n whenever n operations are trimmed. To counter this, we introduce the concept of *Virtual Sequence Numbers (VSN)*. VSN of an entry in a log represents the sequence number the entry would have if the log was never trimmed. Therefore, we must track an *offset* (initially zero) that is updated everytime we trim an OpLog from the top (incremented by the number of entries trimmed). Then to get the VSN of an entry, we can simply add its sequence number to the offset. KnowledgeLogs now store VSNs instead of sequence numbers. While trimming the OpLog, we can trim the corresponding KnowledgeLog entries as well. Just like OpLogs, we track the last entry removed from a KnowledgeLog to indicate the readers the operations that have been trimmed.

Third, if a new replica joins the cluster with an initially empty state, it can force a change in the order of operations. This can be overcome in two ways: we can copy the initial state of the new replica from an arbitrary existing replica, or we can assign an ID to the new replica that is smaller than all the existing replicas. A smaller ID will force the new replica to put its entries at the end of all the existing versions and thus, not alter the current order of operations.

VI. Evaluation

In this section, we empirically evaluate the performance of LSCRDT. We build LSCRDT on top of CSPOT [20], an open-source, distributed runtime system that uses memory-mapped files for its log abstraction. As LSCRDTs combine the advantages of both op-based and state-based CRDTs much like δ-CRDTs [4], we compare the performance of LSCRDTs with that of δ-CRDTs. Note that our goal is not to outperform δ-CRDTs, but to explore the feasibility of LSCRDT while providing all the advantages associated with using logs.

We conduct our experiments using the foundational data types register, counter, and set widely studied in CRDT literature. We use the last-writer-wins (LWW) variant of register and positive-negative (PN) variant of counter for both δ-CRDT and LSCRDT. We use two-phase (2P) variant of set for δ-CRDT

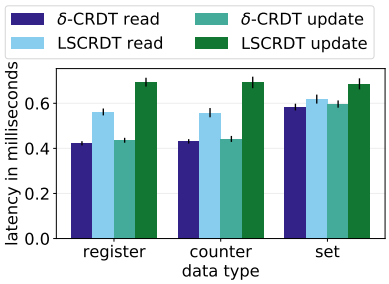


Fig. 4: Comparison of latency between LSCRDTs and δ-CRDTs on operations executed at a single replica.

(an element cannot be added again once removed) whereas LSCRDT set works in the conventional way. We follow the δ-CRDT implementation of [27] for the above-mentioned data types and extend it by adding persistent storage and methods to communicate among devices. we implement δ-CRDT in C and use memory-mapped files for persistent storage.

We focus on four questions in our results: (i) how much extra time is introduced for a single operation to execute due to the introduction of logs to store the data types, (ii) what is the effect of logs on scalability, (iii) how quickly do the merge steps converge, and (iv) how time consuming is versioned read compared to reading the latest value. We run each experiment ten times and show the average result. We reset each data type to its initial state (registers initialized to null, counters initialized to zero, sets initialized to empty set) before each run.

We perform the replication experiments with a cluster of three replicas and one client. Each replica maintains a pool of five worker threads, one among them assuming the role of a writer at a time. All of the machines are running the CentOS 7 Linux operating system each with two dedicated 2GHz vCPUs and 2GB of memory. The machines communicate among themselves using 1Gb/second Ethernet. The average latency among the replicas is observed to be 0.35ms whereas the average latency between any replica and the client is observed to be 0.45ms. In the case of LSCRDTs, each replica executes a merge step with another replica in round-robin fashion at one second intervals. In case if δ-CRDTs, deltas are propagated immediately after local execution of an update operation.

A. Single Node Latency

To evaluate latency, we send 10000 randomly generated operations sequentially from the client to a replica and measure the average latency. The workload contains half reads and half updates. The arguments of the update operations are randomly generated integers between 1 and 10000. In cases where there can be more than one type of update operation, the type is selected randomly. Read operations read the latest versions of respective data types in the case of LSCRDT. All other replicas are inactive for this experiment (i.e. there is no merge step).

Figure 4 shows the read and update latencies for the three data types. In the case of register, the read latency of LSCRDT is 1.3 times that of δ-CRDT. This is expected, as LSCRDT requires two log operations to retrieve the latest

value: one for finding the latest sequence number and one for retrieving the entry at the latest sequence number. On the other hand, δ-CRDT requires a single file read. The write latency of LSCRDT register is 1.6 times that of δ-CRDT register. LSCRDT requires two writes on two different logs for updating a register: one on the OpLog and another one on a KnowledegLog. In the case of counter, the read and update latencies for LSCRDT are 1.3 times and 1.6 times that of δ-CRDT respectively as well. This is expected, as counter and register involve a similar number of log read/write. The current state of a set is the composition of its elements, whereas the current state of a register or counter is a single value. This inherent structural difference results in a higher read latency in the case of sets when compared to the other two data types. Due to the caching of the latest operations in LSCRDT set and checkpointing of elements in the set, the difference between the read/update latencies of δ-CRDT and LSCRDT is small, LSCRDT being 1.1 times slower for reads and 1.15 times slower for writes.

B. Scalability

To evaluate the scalability of LSCRDT, we randomly generate workloads of 10000 operations. As updates are more expensive than reads in general, we vary the percentage of update (read counts shown in parentheses) operations among $1(99)$, $25(75)$, and $50(50)$ to observe the impact of workloads with different update/read composition on scalability. We also vary the number of replicas the client sends requests to among 1, 2, and 3. All 3 replicas are live and perform merge step (LSCRDT) or join (δ-CRDT) even if the client sends requests to fewer than 3 replicas. A client evenly distributes operations across replicas using round-robin without delay.

Figure 5a shows the throughput of the system in operations per second. For LSCRDT registers, throughput increases more than 1.6 times as the number of replicas increases for all workloads. This increase is more than 2.6 times when the number of replicas is increased from 1 to 3 for workloads with 25% and 50% updates. The increase is lower for 1% updates. The increase in throughput with the increase in the number of replicas indicates that LSCRDT registers are scalable, although this increase is not strictly linear. This is due to the processing required for background merge steps. Figure 5b and Figure 5c reveal a similar increase in throughput for counter and set, respectively, as the number of replicas increases.

Figure 6 compares LSCRDT throughput with that of δ-CRDT for the data types using three replicas. The throughput of δ-CRDT data types are slightly better, ranging from 1 to 1.3 times that of LSCRDT data types. As merge steps are expected to be expensive in LSCRDT due to log rollbacks and our system does not support concurrent writes at the same replica, we would expect the throughput to take a hit. We are able to limit this overhead by minimizing the duration during which updates cannot take place to only when the new order of operations is recorded. Specifically, we continue to service update requests during the conflict detection stage of the merge step. To allow reads to be serviced even during conflict resolution, we create the ordered list of operations on a backup log and replace the active log with the backup at the end of ordering all the operations. This design helps in maintaining a high throughput, although the time to converge might still be high as we investigate next.

C. Time to Achieve Stability

To evaluate the impact of the merge step on execution time, we measure the time for the system to reach stability when no new updates are performed. We say a system is stable after a collection of update operations if reads from all the replicas return the same value reflecting the last executed update operation.

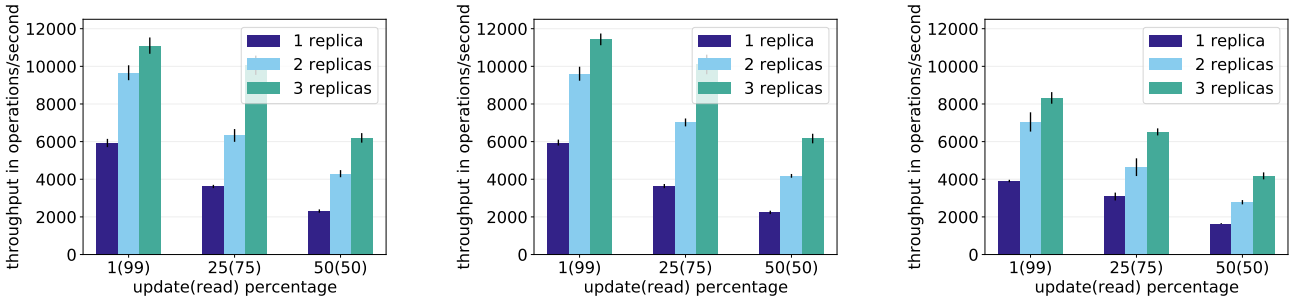
For this experiment, the client evenly distributes 300, 900, and 1500 update requests to replicas using round-robin without delay. Each replica starts a merge step/join after executing all its update operations: 100, 300, and 500 operations for the workloads containing 300, 900, and 1500 operations, respectively. We define T to be the delay between the time the client received the last response of the executed update commands and the time it received a consistent final result for read requests from all replicas.

We denote T values for δ-CRDTs and LSCRDTs by T_δ and T_{LS}, respectively. Table I shows the results. T_{LS} values are around 8 to 12 times the corresponding T_δ values. Deltas can be propagated as soon as an operation is executed locally. The corresponding joins are simple and do not involve rollbacks. On the other hand, merge steps in LSCRDTs possibly involve rollback of logs and re-execution of operations. This is the cost we incur for maintaining a consistent version history and supporting arbitrary non-commutative data types. Note that the reason we do log rollbacks is to support arbitrary non-commutative data type and to maintain a consistent global version history of a data type. If the underlying data type is commutative and we only wish to preserve a local version history, we can avoid rollbacks altogether.

The data from Table I also shows that as the number of total operations increases 3 times and 5 times, the time to reach stability increases proportionally to that. This suggests that LSCRDTs would converge faster in systems with lower update frequencies as the number of updates between consecutive merge steps will be lower. If update requests come in bursts of n operations, T would roughly remain similar. The lower the value of n, the faster the time to reach stability. As LSCRDT allows reads during updates but does not allow multiple updates (and merge is a form of update), making the merge frequencies arbitrarily small would have an adverse effect: it would make the system unavailable for update for more frequent bursts of time.

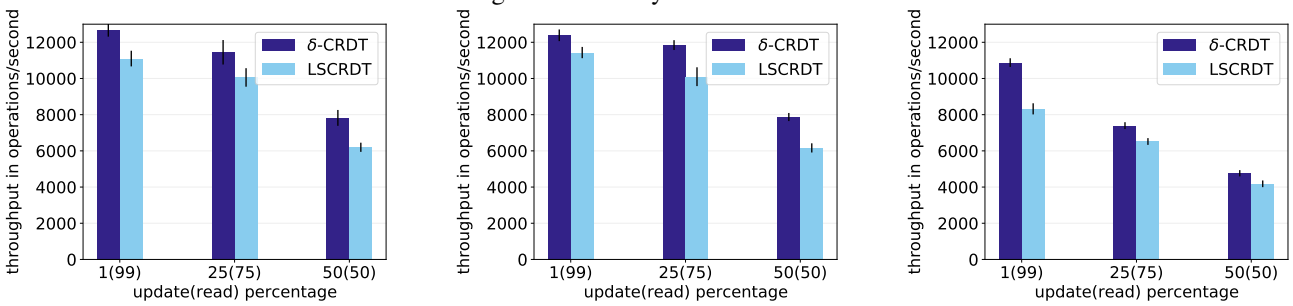
D. Versioned Reads

Maintaining version history is a unique feature of LSCRDT s not available in CRDTs. To understand the overhead associated with querying earlier versions, we send 1000 update requests to a data type. At the end of the update requests, we



(a) Register. (b) Counter. (c) Set.

Fig. 5: Scalability of LSCRDTs.



(a) Register. (b) Counter. (c) Set.

Fig. 6: Comparison of throughput between LSCRDTs and δ-CRDTs using three replicas.

TABLE I: Delay between the last response of a set of update requests and a consistent read from all three replicas.

Data Type	#Updates	T (milliseconds)		T_{LS}/T_{δ}
		T_{δ}	T_{LS}	
Register	300	44.522	548.823	12.327
	900	146.559	1665.450	11.364
	1500	248.129	2849.163	11.483
Counter	300	49.266	636.418	12.918
	900	162.972	1748.149	10.727
	1500	243.693	3098.375	12.714
Set	300	78.649	654.255	8.319
	900	238.207	1792.298	7.524
	1500	389.821	3048.765	7.821

send two sets of 1000 read requests: the first set contains non-versioned reads (i.e. reads to the latest version) whereas the second set contains versioned reads, one read per version.

We show the average latency of the non-versioned and versioned reads in Figure 7. The latencies for versioned and non-versioned reads for both counters and registers are similar. This is expected, as irrespective of the type of read LSCRDT requires the same number of log access.

However, set presents us with a different scenario. Checkpoint logs are accessed for both versioned and non-versioned reads for set. The latest operations are cached in memory whereas non-versioned reads require reading the OpLog as described in Section III-C. The average read latency of non-versioned read on set is 0.7ms whereas that of versioned read on set is 4.5ms, i.e., versioned read is more than six times expensive on average for a $cp\_interval$ value of 100 and the input workload.

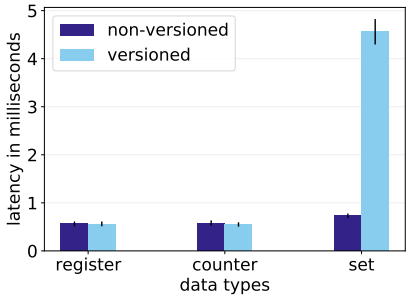


Fig. 7: Comparison of average latency in milliseconds between 1000 non-versioned read (i.e. reads to the latest version) and 1000 versioned read (one read per version) at the end of the execution of 1000 update operations.

VII. Conclusions and Future Work

In this work, we introduce LSCRDTs to integrate the benefits of distributed causal logging into replicated, operation-based CRDTs. By doing so, LSCRDT is the first CRDT system to provide a robust and uniform way to reverse operations for arbitrary data types. In addition, LSCRDT overcomes the restrictions of commutative data types, exactly-once causal delivery, operation idempotence, and data type specific join operations (a side effect of state-based CRDTs). Finally, LSCRDT is the first CRDT system to track version histories of data structures and provide programmatic access to them, while bounding log size.

We design these features and implement them for an open source distributed runtime system that exports an efficient distributed log abstraction. We define a combination of logs

that together provide these features and show how their use can be optimized in various ways to keep their overhead low. We empirically evaluate and analyze this overhead by comparing operation latency and throughput of LSCRDT to that of δ-CRDT. Our results show that LSCRDT introduces 1.1-1.6x operation latency, 1.3x throughput overhead, and similar scalability and stability characteristics across workloads and data types. Moreover, our results show that the latency of versioned read and non-versioned reads for simple data types is similar, but higher for complex data types (e.g. sets). In future work, we will investigate additional optimizations and caching strategies, as well as log-augmentation for state-based CRDTs.

REFERENCES

[1] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[2] N. Preguiça, C. Baquero, and M. Shapiro, *Conflict-Free Replicated Data Types CRDTs*. Cham: Springer International Publishing, 2019, pp. 491–500. [Online]. Available: https://doi.org/10.1007/978-3-319-77525-8_185

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," 2011.

[4] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 162–173, 2018.

[5] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1–7.

[6] W.-T. Lin, F. Bakir, C. Krintz, R. Wolski, and M. Mock, "Data repair for Distributed, Event-based IoT Applications," in *ACM International Conference on Distributed and Event-Based Systems*, 2019.

[7] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[8] S. Weiss, P. Urso, and P. Molli, "Logoot-undo: Distributed collaborative editing system on p2p networks," *IEEE transactions on parallel and distributed systems*, vol. 21, no. 8, pp. 1162–1174, 2010.

[9] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils, "Lseq: an adaptive structure for sequences in distributed collaborative editing," in *Proceedings of the 2013 ACM symposium on Document engineering*, 2013, pp. 37–46.

[10] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354–368, 2011.

[11] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 2009, pp. 395–403.

[12] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for p2p collaborative editing," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, 2006, pp. 259–268.

[13] M. Kleppmann and A. R. Beresford, "A conflict-free replicated json datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, 2017.

[14] M. Zawirski, C. Baquero, A. Bieniusa, N. Preguiça, and M. Shapiro, "Eventually consistent register revisited," in *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, 2016, pp. 1–3.

[15] S. Dolan, "Brief announcement: The only undoable crdts are counters," in *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020, pp. 57–58.

[16] G. Younes, P. S. Almeida, and C. Baquero, "Compact resettable counters through causal stability," in *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, 2017, pp. 1–3.

[17] W. Yu and S. Rostad, "A low-cost set crdt based on causal lengths," in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020, pp. 1–6.

[18] R. Brown, Z. Lakhani, and P. Place, "Big (ger) sets: decomposed delta crdt sets in riak," in *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, 2016, pp. 1–5.

[19] Facebook, "LogDevice," 2020, https://engineering.fb.com/core-data/logdevice-a-distributed-data-store-for-logs/ Accessed 29-Feb-2020.

[20] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, "CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT," in *ACM Symposium on Edge Computing*, 2019.

[21] Basho, "Bitcask," https://docs.riak.com/riak/kv/2.2.3/setup/planning/backend/bitcask/index.html.

[22] K. De Porre, F. Myter, C. De Troyer, C. Scholliers, W. De Meuter, and E. G. Boix, "Putting order in strong eventual consistency," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2019, pp. 36–56.

[23] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 265–278.

[24] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 385–400.

[25] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 309–324.

[26] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, "Putting consistency back into eventual consistency," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–16.

[27] C. Baquero, "Delta-enabled crdts," https://github.com/CBaquero/delta-enabled-crdts, 2015.