Log-Based CRDT for Edge Applications

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

***Abstract*—In this paper, we investigate extensions for Conflict-Free Replicated Data Types (CRDTs) that permit their use in failure-prone, heterogeneous, resource-constrained, distributed, multi-tier (cloud/edge/device) cloud deployments such as the Internet-of-Things (IoT), while addressing multiple CRDT limitations. Specifically, we employ distributed logging to implement robust, strong eventual consistency of replicas. Our approach also enables uniform reversal of operations and precludes the requirement of exactly-once delivery and idempotence imposed by operation-based CRDTs. Moreover, it exposes CRDT versions for use in debugging and history-based programming. We evaluate our approach for commonly used CRDTs and show that it enables higher operation throughput (up to 1.8x) versus conventional CRDTs for the workloads we consider.**

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

I. Introduction

The present cloud computing paradigm is becoming increasingly ubiquitous and often employs a multi-tier (cloud/edge/device) architecture. Many modern conventional cloud deployments not only require efficient computation and storage solely on the cloud tier, but also seamless integration of computational/storage concepts on edge and device tier. A prime example of these are edge applications, where the edge servers must communicate with high-end cloud servers while overseeing Internet of Things (IoT) devices at the same time. Unfortunately, the same technologies used to guarantee failure resilience and robustness on the cloud do not translate directly to the edge/device layer due to resource-constraints and unstable network connectivity. Thus, new programming support and data abstractions are needed to manage this complexity and automatically enhance the robustness and efficiency of multi-scale cloud deployments.

Toward this end, we introduce Log-Structured CRDTs (LSCRDTs) – a new form of *C*onflict-Free *R*eplicated *D*ata *T*ypes [1]–[3] that is better suited for use in multi-tier cloud deployments than its antecedents. In general, CRDTs enable program data types to be distributed, shared (concurrently modified), replicated, and made consistent (via clever update merging). Specifically, CRDT replicas employ Strong Eventual Consistency (SEC) [1], which guarantees that two replicas reach the same state if they receive the same set of updates (possibly in different orders). These features facilitate CRDT robustness, high availability, and coordination avoidance [4], which makes them suitable for failure prone settings.

However, many existing CRDT designs impose limitations that make them difficult to use in multi-tier cloud deployments. In particular, previous CRDT formulations do not support *operation reversal* – the process of reverting a data type to a previous state for operations that are not executed.[1] They do not easily support non-commutative operations, and (for operation-based CRDTs) they cannot tolerate out of order operation delivery.

LSCRDT uses distributed logs to overcome these limitations. As logs are append-only, our approach is able to use them to further reduce the coordination required to merge inconsistent replicas. LSCRDT guarantees that replicas execute operations in the same order, irrespective of data type thus removing the need for commutability. LSCRDTs also avoid the complex networking required to guarantee exactly-once, causal delivery of operations required by operation-based CRDTs. Finally, LSCRDT provides programmatic access to data structure versions, which is useful for debugging, history-based programming [6] and data replay and repair [7].

In this paper, we describe the design and implementation of LSCRDT and provide a comparative study of LSCRDT and δ-CRDT [8] – a popular category of CRDT that combines the advantages of a wide range of CRDT implementations. We evaluate latency and throughput using three, extensively studied, CRDT data types – register, counter, and set [9]–[13]. Our results show that, apart from providing the aforementioned properties, LSCRDT outperforms δ-CRDT in terms of write latency. Moreover, for update-heavy workloads (typical of sensor-driven edge applications), LSCRDT exhibits up to $1.8x$ higher throughput than δ-CRDT.

II. Related Work

Conflict-Free Replicated Data Types (CRDTs) are abstract data types that provide a principled approach for the asynchronous reconciliation of divergent data resulting from concurrent updates [1]. CRDTs provide strong eventual consistency (SEC), which 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 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

[1] Note that operation reversal is distinct from the *undo* operation [5], which may have side effects.

communicate state, they require exactly-once causal broadcast. Delta State CRDTs (δ-CRDTs) [8] combine the advantages of state-based and op-based CRDTs. Like 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 op-based approach, they do not require that the full replica state be communicated, but rather, they communicate only state changes or "deltas".

Most CRDT studies are evaluated using the foundational data types, register, set, and counter [9]–[15]. CRDTs for JSON data [16] allow programmers to create custom data types by nesting maps and lists into new CRDTs. Although this approach adds a new dimension to the foundational data types, it is still insufficiently generic (e.g., we can not create a self-balancing tree with JSON). Other works discuss replicated trees capable of executing arbitrary concurrent *move* operations [17], where a log of move operations is maintained, possibly involving rollback and replay. As we describe in Section VI, LSCRDT uses a similar approach. Strong eventually consistent replicated objects (SECROs) [18] are general-purpose data types that implements SEC without placing any restriction on operations, i.e., operations can be non-commutative. SECROs find a conflict-free ordering of concurrent operations depending on application-specific information. However, to ensure that the application-specific conditions are not violated, SECROs at times must give up on a computation path and backtrack to a previous state, thus imposing significant computational load.

III. Motivation

In this section, we motivate the integration of logs and general purpose, operation-based CRDTs for use at the edge. Many edge applications do not require strong consistency (e.g., smart locks [19], energy management [20], temperature prediction [21], etc.) and can choose a weaker consistency model in favor of increased availability, lower coordination overhead, and better energy efficiency. Due to their ability to provide these advantages, CRDTs are suitable for many edge deployments. Although CRDTs are not new, they have been employed mostly for collaborative editing [22]–[26]. Edge deployments impose new environmental characteristics that past research does not fully address. Specifically, edge applications often coordinate among devices deployed in an environment with poor and unreliable network connectivity. They consist of heterogeneous, resource-constrained devices (with regards to computation, space, and power). Moreover, data-driven applications also demand data resilience, availability, and the ability to audit data lineage. Finally, the ubiquity of edge applications makes a generalized abstraction approach to storing and manipulating data desirable [27].

Past works attempt to address different aspects of these challenges, but none provide a holistic approach within a single system. For example, δ-CRDT can tolerate unreliable networking but does not provide a general approach to merge/join deltas (e.g., joining deltas for counters is different than that for sets). Moreover, in many cases, δ-CRDTs must resort to using a variant of the data type rather than the original one (e.g., using a 2P set instead of a conventional set).

LSCRDT can both tolerate unreliable networking and manage arbitrary data type without the need for data type-specific join algorithm. In LSCRDT, each replica logs the executed operation along with a unique *version stamp* (allowing detection of duplicates) in causal order. Each replica periodically reads from other replicas' logs in log order (e.g. similar to what is done in Kafka [28] logs: each replica reads monotonically increasing offsets). LSCRDT maintains causal order by ensuring log order is maintained during reads.

Logs also provide immutability, which can be used to aid data lineage tracking and versioning [29], data availability and robustness, and application debugging. For example, efficient versioning is critical for temporal queries (typical in stream-based edge applications), and for operation reversal and rollback, and replay of applications and analyses, in the face of errors and malfunctioning sensors and services (which are common occurrences in large scale multi-tier IoT deployments) [7]. Moreover, logs aid interoperability of heterogeneous devices via the use of a simple interface.

Finally, our LSCRDT design is operation based to facilitate generality (i.e. to be agnostic to the data structure type and operation implementation). Many CRDTs require *type-specific* merge/join algorithms [2]. LSCRDT instead forms a consistent order of operations, irrespective of the underlying data type. As we will see in Section VI, agreeing upon a consistent order of operation is not type-specific in LSCRDT, making it more generic and extensible than existing approaches.

IV. 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 upon recovery.

Over time replicas may diverge from each other due to update requests from clients (i.e., 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 (i.e. not previously executed at the reader) 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 LSCRDT merge step in Section VI. Note that during a merge step, a reader 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 Raft [30].

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 determined from the timestamped operations of the source as described in Section VI. 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.

Various algorithms have been proposed to maintain order in list or sequence CRDTs such as Logoot [22], LSEQ [23], RGA [24], Treedoc [31], and WOOT [32]. Our method to maintain order among logs of operations is an adaptation of [16], which is based on RGA [24]. 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 [16], the use of logs introduces a new performance challenge – avoiding log scans. We explain how we can avoid full log scans in Section VI.

V. Data Types Using Logs

We next explain how LSCRDTs are stored using logs. We overview the replication process in Section VI. We choose three data types widely studied in CRDT literature for this exposition: registers, counters, and sets [9]–[13]. 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 [28], *LSN*s in the case of Facebook LogDevice [33], and *sequence number*s in the case of CSPOT [34]). 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 [35]), 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 its OpLog. For example, suppose a request to assign the value 5 to a register comes to X_A. 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, X_A simply returns the *val* field of the last entry in its OpLog. If some previous version stamp vs_i is supplied as an argument of value, the replica can search the OpLog for 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 [36]. 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 (1*A*, *inc*, 5, 5), (2*A*, *dec*, 2, 3) and (3*A*, *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

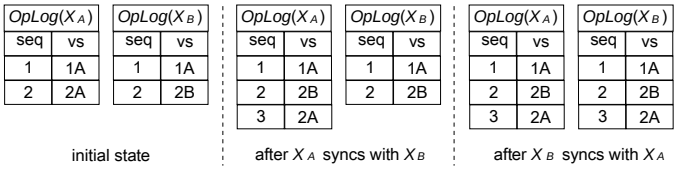
Our set data type supports *add* and *remove* update operations and *in* and *all* read operations. Note that although CRDTs resort to using some variant of sets such as two-phase set (2P-set [1]), grow-only set (G-set [2]) etc; LSCRDT set works like a conventional set due to its capability to find a 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 up to 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 every time 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. A version of a data type might change due to updates from other replicas (cf. Section VI). In that case, the checkpoint that contains that version and the following checkpoints must be overwritten.

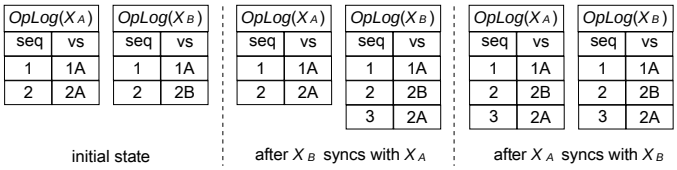
VI. Merge Step

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 total order on the execution of operations [18]. An alternative to imposing a total order for arbitrary non-commutative data types is to switch between stronger and weaker form of consistency [37]–[40]. However, as discussed in Section IV, 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 [16] which is based on RGA [24] for maintaining order in the list i.e. the OpLog, as explained next.

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 $insert(vs_{new}, vs_{pred})$.



(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.

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 1*A* 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 2*A* and 2*B* 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 VI-B how full log scans can be avoided). Both OpLogs have 1*A* as the first entry, so no action is needed. However, X_B has 2*B* in the second entry whereas X_A has 2*A*. This is equivalent to the insert operation $insert(2B, 1A)$ i.e. insert 2*B* after 1*A* in $OpLog(X_A)$ (as 2*B* comes after 1*A* in $OpLog(X_B)$). We note how the insertion operation is implicitly embedded in the log order. As X_A currently has 2*A* after 1*A* and $2B > 2A$, it can place 2*B* after 1*A*. "Placing 2*B* after 1*A*" is a multi-step process: X_A trims its OpLog up to sequence number 1, appends the entry containing 2*B*, and finally re-appends the entry containing 2*A*. 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 $2A < 2B$, $2A$ 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.

Note that we could have forgone this relatively complex ordering following [16] and instead chosen a strict ascending order of version stamp counters, breaking ties through lexicographical order of node IDs. However, this approach would have resulted in interleaving sequences of operations made by different replicas concurrently. Our current choice of the method in [16] on the other hand makes sure concurrent sequence of operations executed by different replicas are not interleaved. Also note that to break tie between concurrent operations we choose the greater version stamp to take precedence over the smaller one (e.g. $2B$ appears before $2A$) to maintain similarity with existing work [16]. In practice, we could have chosen the reverse.

Proof of convergence of $insert$: To understand how $insert$ 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.

We next introduce a new log that helps us to avoid full log scan (Section VI-A). We show how the conflict detection stage uses this log to detect the presence and point of conflict (Section VI-B). Section VI-C then describes how the conflict resolution stage takes this 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. Thus, 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. For improved performance, we can also opt to cache a fixed number of entries from the end of KnowledgeLog in memory. As all the information needed to maintain a KnowledgeLog are present in the OpLog, KnowledgeLogs can be reconstructed after a system crash.

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

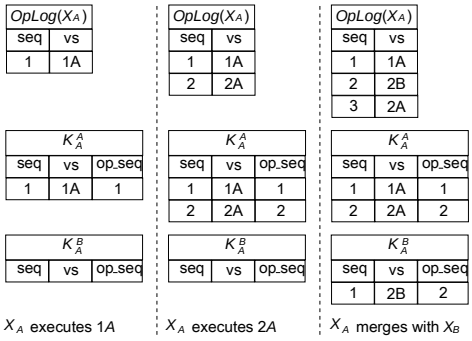


Fig. 2: Mapping from version stamps to sequence number of OpLog in KnowledgeLogs.

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.

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 starts 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.

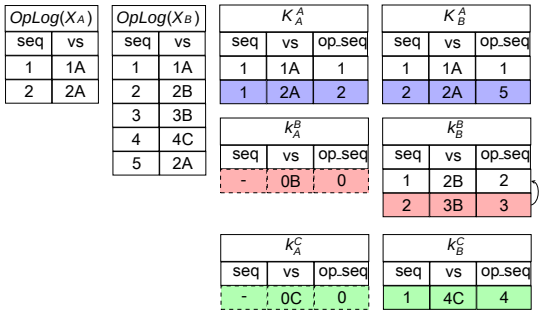


Fig. 3: OpLogs and KLogs of replicas X_A and X_B in a system with three replicas. Dashed entries represent placeholders used during computation when a knowledge log is empty. During conflict detection, the reader X_A compares the same colored entries with each other to find the earliest possible point 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.

Proof that a reader can identify all operations unknown to it during conflict detection: Reader X_i will identify all the operations unknown to it if there are no operations in $OpLog(X_j)$ before the sequence number $source_{start}$ that X_i does not know of (as X_i starts reading $OpLog(X_j)$ from the sequence number $source_{start}$). We can prove this by contradiction. Let us assume there is indeed a version stamp vs in $OpLog(X_j)$ at sequence number $source^0_{start}$ such that $source^0_{start} < source_{start}$ and $vs.nodeID = q$. That would mean one of the following: (i) $p \neq q$. That is, the tail of K_j^q 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_j^p with vs greater than $tail(K_i^p).vs$ 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_j^p that X_i has not seen. Hence we arrive at a contradiction and the reader must be able to identify all operations unknown to it during conflict detection.

Let the version stamp of the sequence number $source_{start} - 1$ in $OpLog(X_j)$ be 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's version stamp is 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_A^m and K_B^m, $m \in \{A, B, C\}$, 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 $tail(K_B^C)$ (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 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.

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 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 $OpLog(X_i)$.

VII. 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 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 due to 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 VI 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 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 user-defined number ($> K$) of versions have been recorded. 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 entries that can be trimmed. Instead, it consults the tails of KnowledgeLogs. 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 in KnowledgeLogs will be off by n whenever n operations are trimmed. To counter this, we introduce *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 every time 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 notify the readers of the trimmed operations. *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.

VIII. Evaluation

In this section, we empirically evaluate the performance of LSCRDT. As LSCRDTs combine the advantages of both op-based and state-based CRDTs much like δ-CRDTs [8], we compare the performance of LSCRDTs with that of δ-CRDTs. We implement both LSCRDT and δ-CRDT in C and use memory-mapped files for persistent storage, including logs. 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 (an element cannot be added again once removed) whereas LSCRDT set works in the conventional way. We follow the δ-CRDT implementation of [41] for the data types.

In our results we focus on: (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, and (iii) 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 before each run. We perform the replication

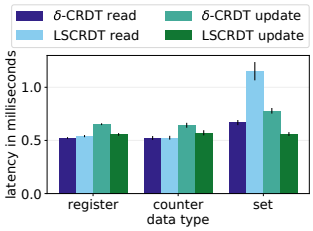


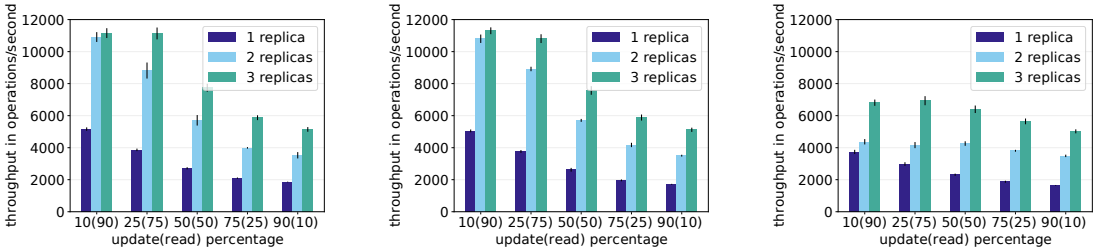
Fig. 4: Comparison of latency between LSCRDTs and δ-CRDTs on operations executed at a single replica.

experiments with a cluster of three replicas and one client. Many cloud-based storage systems such as Amazon S3 use three nodes for replication. Although edge/device tier do not have the same level of reliability as cloud servers, it is not uncommon for edge applications to have a replication factor of three as well [19], [34], [42]. 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 machines 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 queued for propagation immediately after local execution.

A. Single Node Latency

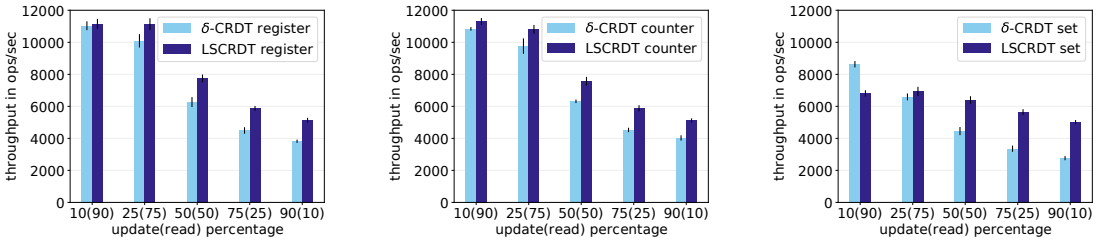
To evaluate latency, we send 10000 randomly generated operations (half reads, half updates) sequentially from the client to a replica and measure the average latency. The arguments of the update operations are randomly generated integers between 1 and 1000. Read operations read the latest versions of respective data types in the case of LSCRDT.

Figure 4 shows the read and update latencies for the three data types. In case of all three data types, LSCRDT shows a lower update latency than δ-CRDT counterparts. Specifically, LSCRDT is $1.17x$, $1.12x$, and $1.39x$ faster than δ-CRDT for register, counter, and set respectively. There is no significant difference in read latencies between LSCRDT and δ-CRDT for register and counter. However, δ-CRDT set provides faster read than LSCRDT set. Specifically, the read latency of δ-CRDT set is 0.668ms whereas that of LSCRDT set is 1.151ms. That is, δ-CRDT reads are 1.72 times faster than LSCRDT for set. This difference in read latency is expected. Recall that although LSCRDT set works in the conventional way, the variant of δ-CRDT set we are using (i.e., 2P-set) does not allow an element to be added again once it has been removed. That is, the query in 2P-set can be performed faster by first looking at the list of elements marked as removed. On the other hand, LSCRDT set has to first read the last checkpointed state from a log and take into account all the operations that has been performed since that checkpoint. Although the read latency of LSCRDT set is higher than that of δ-CRDT set, we will see in Section VIII-B that the former has higher throughput in the presence of high volume of updates. As devices at the edge are generally write heavy, LSCRDT is thus a better option than δ-CRDT for edge applications.



(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.

We also perform experiments to compare the performance of versioned reads and non-versioned reads. Our results show that for register and counter there is no significant difference in read latency. However, versioned reads (1.29ms) for LSCRDT set is 1.1x slower than non-versioned reads (1.17ms). This is expected, as non-versioned reads can leverage the latest operation cached in memory, whereas versioned reads might require reading the OpLog as described in Section V-C.

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) operations among 10(90), 25(75), 50(50), 75(25), and 90(10) 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 (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 5 shows the throughput of the system in operations per second. For LSCRDT registers, throughput nearly doubles as the number of replicas is increased from one to two for most workloads. This increase is around 2.8 times when the number of replicas is increased from 1 to 3 for all workloads except that with 10% 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 reveals a similar increase in throughput for counter as the number of replicas increases. We can observe a similar trend in increase in throughput for set for workloads with a higher percentage of update from Figure 5c. The increase in throughput is not as pronounced for workloads with a lower percentage of update due to the higher read latency of LSCRDT set.

Figure 6 compares LSCRDT throughput with that of δ-CRDT for the data types using three replicas. As the percentage of update is increased, LSCRDT data types start showing higher throughputs than δ-CRDT counterparts. Specifically, LSCRDT register and counter show 1.1x, 1.2x, 1.3x, and 1.3x higher throughput than δ-CRDT counterparts for workloads with 25, 50, 75, and 90 percentage of update respectively. In the case of LSCRDT set, the increase in throughput is 1.1x, 1.4x, 1.7x, and 1.8x than its δ-CRDT counterpart for workloads with 25, 50, 75, and 90 percentage of update respectively. Note that LSCRDT set has a higher read latency but a lower write latency than δ-CRDT set. The lower write latency along with the efficient lock-free merge step of LSCRDT results in a higher throughput in LSCRDT set for workloads with a high volume of updates. As edge applications are typically write-heavy, LSCRDT presents itself as the better option.

IX. Conclusion

In this work, we introduce LSCRDTs to integrate the benefits of distributed causal logging into operation-based CRDTs. By doing so, LSCRDT can 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. Our results show that LSCRDT can result in up to 1.8x higher throughput than δ-CRDT, making it suitable for update-heavy edge workloads.

References

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

[2] N. Preguic¸a, 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. Preguic¸a, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," Inria, Tech. Rep. RR-7506, 2011.

[4] P. Helland, "Immutability changes everything," in *Conference on Innovative Data Systems Research*, 2015, http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf Accessed 15-Sep-2019.

[5] W. Yu, V. Elvinger, and C.-L. Ignat, "A generic undo support for state-based crdts," in *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[6] D. Meissner, B. Erb, F. Kargl, and M. Tichy, "Retro-lambda: An event-sourced platform for serverless applications with retroactive computing support," in *Intl. Conf. on Distributed and Event-based Systems*, 2018.

[7] 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.

[8] 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.

[9] M. Zawirski, C. Baquero, A. Bieniusa, N. Preguic¸a, 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] V. Enes, P. S. Almeida, C. Baquero, and J. Leitão, "Efficient synchronization of state-based crdts," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 148–159.

[15] J. Bauwens and E. G. Boix, "Improving the reactivity of pure operation-based crdts," in *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, 2021, pp. 1–6.

[16] 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.

[17] M. Kleppmann, D. P. Mulligan, V. B. Gomes, and A. R. Beresford, "A highly-available move operation for replicated trees and distributed filesystems," 2020.

[18] 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.

[19] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity internet of things devices," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, 2016, pp. 461–472.

[20] M. I. Naas, L. Lemarchand, P. Raipin, and J. Boukhobza, "Iot data replication and consistency management in fog computing," *Journal of Grid Computing*, vol. 19, no. 3, pp. 1–25, 2021.

[21] N. Golubovic, R. Wolski, C. Krintz, and M. Mock, "Improving the accuracy of outdoor temperature prediction by iot devices," in *2019 IEEE International Congress on Internet of Things (ICIOT)*. IEEE, 2019, pp. 117–124.

[22] 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.

[23] 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.

[24] 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.

[25] X. Lv, F. He, Y. Cheng, and Y. Wu, "A novel crdt-based synchronization method for real-time collaborative cad systems," *Advanced Engineering Informatics*, vol. 38, pp. 381–391, 2018.

[26] 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*, 2016, pp. 259–268.

[27] V. A. Barros, J. C. Estrella, L. B. Prates, and S. M. Bruschi, "An iot-daas approach for the interoperability of heterogeneous sensor data sources," in *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2018, pp. 275–279.

[28] 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.

[29] N. Saquib, C. Krintz, and R. Wolski, "Pedals: Persisting versioned data structures," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 179–190.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

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

[37] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguic¸a, 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.

[38] 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.

[39] 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.

[40] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguic¸a, 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.

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

[42] N. Golubovic, R. Wolski, C. Krintz, and M. Mock, "Improving the Accuracy of Outdoor Temperature Prediction by IoT Devices," in *IEEE Conference on IoT*, 2019.