2013 IEEE 33rd International Conference on Distributed Computing Systems

A Scalable Conflict-free Replicated Set Data Type

Andrei Deftu
1&1 Internet AG
andreideftu@gmail.com

Jan Griebsch
1&1 Internet AG
jan.griebsch@1und1.de

***Abstract*—Replication of state is the fundamental approach to achieve scalability and availability. In order to maintain or restore replica consistency under updates, some form of synchronization is needed. *Conflict-free Replicated Data Types* (CRDTs) ensure eventual consistency, such that replicas converge to a common state, equivalent to a correct sequential execution without foreground synchronization.**

A particular CRDT is the *set* data type, which is a pervasive abstraction for storing collections of unique elements and constitutes an important building block for other, more complex data structures. Since the original specification is not scalable, we improve it by introducing an efficient algorithm for sending deltas of updates between replicas and by partitioning a set replica into disjunctive subsets. We further add support for limited-lifetime elements, which, in turn, enable simple garbage collection strategies to address the problem of unbounded database growth. Lastly, implementation details and evaluation results of a client library for this data structure are presented.

***Keywords*-eventual consistency; data replication; distributed systems**

I. Introduction

The common denominator of current commercial Internet services ("cloud") is the promise and the demand to make data available, from anywhere, at any time, with low latency. Now, features like throughput, consistency, and fault-tolerance are necessities, not optimizations, and are included in the design of any modern distributed data system from the beginning. One cannot accept delays in database requests of more than a few hundred milliseconds or downtimes of even a few minutes. The trend is clear: we need to process more data, quicker, and without interruptions.

Replication of data has been the pattern to address fault-tolerance on one hand, while providing the means for achieving higher scalability and performance on the other hand. However, it introduces the problem of maintaining or restoring replica consistency - understood here as replicas behaving identically to requests - under concurrent updates and failures. CAP is a well known theorem [1] which states that any distributed computer system cannot provide simultaneous guarantees for the aforementioned requirements. The majority of the current Internet services prefer availability and partition tolerance, while accepting a weaker form of consistency. The choice has the advantage of lower latencies for client requests and higher scalability, but achieving consistency between replicas still remains an open issue.

One attractive approach is to provide *eventual consistency* [2], [3], which allows any replica to apply updates locally, while the operations are later sent asynchronously to all the others. In this way, all replicas eventually apply all updates, possibly even in a different order. With this weaker form of consistency, considered acceptable for some applications, data remains available when the network is partitioned. The downside is that a complex background consensus algorithm for reconciling conflicting updates is generally needed [4], which makes current approaches ad-hoc and error-prone. Amazon's shopping cart constitutes a well-known example in this sense [5]. Alternatively, several systems execute an update immediately and later discover that it conflicts with another [4]. So they roll-back to resolve the conflict.

Conflict-free Replicated Data Types (CRDTs) [6] were designed specifically to solve this problem by employing a new type of consistency, *strong eventual consistency*, as defined in Section II. Replicas of CRDTs are proved to converge in a self-stabilising manner without blocking client operations and without having to deal with consensus, complex conflict resolution, or roll-backs. However, this model imposes some mathematical - and, in consequence, semantic - constraints, that make it unsuited for some data structures or use-cases.

Composites of CRDTs yield the same properties, and thus basic structures, e.g. counters, shared mutable variables or sets can be used as building blocks in forming more complex ones, like maps or graphs. For a practical use-case scenario, consider how an event tracking mechanism can be implemented in order to prevent attacks on an Internet service provider. Filters are used to keep track and to limit the number of events allowed for a given IP address or account, such as login attempts, password changes, emails sent, and so on. A replicated counter can store the number of login attempts from one IP address, while a replicated set the corresponding unique passwords tried. Since this case requires high throughput for writes of runtime data and low latencies for reads, traditional synchronization or conflict resolution are not acceptable. CRDTs are very attractive, as all updates are persistent and can immediately be applied locally at the source replica. Consistency is achieved later, during a background asynchronous phase in which all replicas eventually apply all updates. Furthermore, the composability nature of CRDTs allows this use case to be easily extended to a graph-like structure: store relations among various events and entities, such as account, IP addresses, aliases, and login attempts in order to better detect malicious behaviour with heuristic algorithms. Another application of the CRDTs is cooperative editing [7].

The focus of this paper is an extension of a particular CRDT: the *set* data type as defined in [6]. Therein, the authors define two functionally equivalent synchronizations variants: The *operations-based* synchronization distributes and merges update operations among different replicas, while the *state-based* synchronization does the same with complete replica state(s). Our paper makes the following contributions:

- An improvement to the set type is provided which transmits only incremental state deltas.
- Acknowledging the fact that large data structures cannot be efficiently stored on just one machine, a partitioning scheme is often desired. Sets of elements fit well in this category of structures, being easily partitioned in several disjunctive subsets and distributed across a cluster of machines. This solves both the problem of data growth by achieving higher scalability and the problem of performance bottleneck by sharing the load. Thus, a second contribution is an extension to the set specification to support per-replica partitioning capability, or *sharding*.
- CRDTs usually lead to an increase in database size with each update operation. Also, we want to add a feature for limited-lifetime elements: discarding elements older than a given time value. We discuss an asynchronous garbage collection mechanism which solves both these issues.
- Finally, these concepts are put into practice through the implementation and evaluation of a client library.

II. Conflict-free Replicated Data Types

Achieving consistency is one of the hardest problems in distributed systems. What we ideally want to have is replicas which are *strongly consistent*, in the sense that any update happening at one replica is made instantaneously visible at all the others. Since this approach implies synchronization after each update operation, essentially leading to a serial execution, it is rarely used in practice. *Eventual consistency* [2], [3] is a weaker form that moves the synchronization phase out of the critical path, to the background. In this way, updates can always be made locally, even if the network is partitioned. However, it still requires conflict arbitration techniques, such as a consensus algorithm or roll-backs [4].

Conflict-free Replicated Data Types (CRDTs) introduce the concept of *Strong Eventual Consistency* (SEC) [6]. The idea is to design data structures, such that all updates have a deterministic outcome and thus they can be made immediately persistent. In this way, conflicts are altogether avoided and a consensus or roll-backs are not longer necessary. There are two styles for defining CRDTs.

State-based replication. In this approach, each update operation is executed entirely at the source replica, modifying its state. Subsequently, every replica occasionally sends its local state to some other replica, which *merges* it into its own state. Convergence is achieved by eventually delivering every update directly or indirectly to all replicas. There are well known protocols in the literature that do this in a fault-tolerant manner, such as gossip or anti-entropy [8], [9].

State-based style uses the concept of *semilattice* defined next. A *partially ordered set* is the pair consisting in a set together with a binary relation $\sqsubseteq$ which establishes an order between the elements of the set. A *least upper bound* (LUB) $\sqcup$ is defined as follows: for any elements x, y from the set, $m = x \sqcup y$ is LUB of $\{x, y\}$ if and only if $x \sqsubseteq m$ and $y \sqsubseteq m$ and there is no other $m' \sqsubseteq m$ such that $x \sqsubseteq m'$ and $y \sqsubseteq m'$. From this definition, it follows that a LUB is: i) commutative: $x \sqcup y = y \sqcup x$, ii) idempotent: $x \sqcup x = x$, iii) associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$. A partially ordered set which has a LUB is called a *join-semilattice* [10] (or just *semilattice* from now on). An example of semilattice is $(2^{\{x,y,z\}}, \subseteq)$, where $2^{\{x,y,z\}}$ is the power set of $\{x, y, z\}$ and the LUB is given by set-union.

In order to be SEC, a state-based object, also called a *Convergent CRDT* (CvRDT), needs to fullfill the following conditions:

- Its payload takes values in a semilattice $(S, \sqsubseteq)$.
- Updates monotonically increase the states according to $\sqsubseteq$.
- The *merge* operation computes the LUB between the local and remote states.

An example of state-based object can be one whose payload is an integer value, $\sqsubseteq$ is common the integer order $\leq$, and where $merge() \overset{def}{=} max()$.

Operation-based (op-based) replication. Here, the system transmits only update operations across replicas, and not whole states. Applying an update is split in two phases. The first one, called *prepare-update*, has no side-effects and its role is to compute some intermediary results. When it terminates, the system sends the update operation, its parameters, and possible intermediary results from the first phase to all replicas, including the source. Here, in the second phase, called *downstream* or *effect-update*, the operation is executed immediately at source and asynchronously at all other replicas if a downstream precondition is met[1]. This second phase cannot return results and has to execute atomically.

An update u is said to *happen-before* an update u' at some replica, denoted by $u \rightarrow u'$, if u has been applied when u' executes. Two updates are *concurrent* if no one happens before the other: $u \parallel u' \iff u \not\rightarrow u' \land u' \not\rightarrow u$. The conditions for an op-based object, or a *Commutative CRDT* (CmRDT), to achieve SEC are:

- Related updates by happened-before, $u \rightarrow u'$, are applied in the same order at all replicas: first u and then u'.
- Concurrent operations, $u \parallel u'$, *commute*: executing u immediately followed by u' leads to the same state as executing u' immediately followed by u. Concurrent operations may be delivered in any order.

There are common epidemic protocols, such as Bayou's anti-entropy [9], which guarantee this causal ordering for updates delivery.

Interestingly, these two replication styles are equivalent [6]: any data type that can be implemented as a state-based object can also be implemented as an op-based object and vice versa. However, there are trade-offs which should be considered

[1]An example of such precondition could be to allow the removal of an element from a replicated set only if it is present in the set at source.

Specification 1 G-Set (state-based)

```
**payload** $A = \varnothing$
**update** $add(e)$
    $A := A \cup \{e\}$
**query** $lookup(e)$ : boolean
    **return** $e \in A$
**compare** $(S, T)$ : boolean
    **return** $S.A \subseteq T.A$
**merge** $(S, T)$ : payload
    **return** $S.A \cup T.A$
```

Specification 2 2P-Set (state-based)

```
**payload** $A = \varnothing, R = \varnothing$
**update** $add(e)$
    $A := A \cup \{e\}$
**update** $remove(e)$
    **pre** $lookup(e)$
    $R := R \cup \{e\}$
**query** $lookup(e)$ : boolean
    **return** $e \in A \wedge e \notin R$
**compare** $(S, T)$ : boolean
    **return** $S.A \subseteq T.A \vee S.R \subseteq T.R$
**merge** $(S, T)$ : payload
    **let** $U.A = S.A \cup T.A$
    **let** $U.R = S.R \cup T.R$
    **return** $U$
```

Specification 3 OR-Set (op-based)

```
**payload** $S = \varnothing$
**query** $lookup(e)$ : boolean
    **return** $\exists u : (e, u) \in S$
**update** $add(e)$
    **prepare**$(e)$ : tag
        **let** $\alpha = unique()$
        **return** $\alpha$
    **effect**$(e, \alpha)$
        $S := S \cup \{(e, \alpha)\}$
**update** $remove(e)$
    **prepare**$(e)$ : set
        **pre** $lookup(e)$
        **let** $R = \{(e, u) \mid \exists u : (e, u) \in S\}$
        **return** $R$
    **effect**$(R)$
        **pre** $\forall (e, u) \in R : add(e, u)$ has been delivered
        $S := S \setminus R$
```

Specification 4 OR-Set (state-based)

```
**payload** $A = \varnothing, R = \varnothing$
**query** $lookup(e)$ : boolean
    **return** $\exists \alpha : (e, \alpha) \in A \wedge \nexists \beta : (a, \alpha, \beta) \in R$
**update** $add(e)$
    **let** $\alpha = unique()$
    $A := A \cup \{(e, \alpha)\}$
**update** $remove(e)$
    **pre** $lookup(e)$
    **let** $\beta = unique()$
    $R := R \cup \{(e, \alpha, \beta) | \exists (e, \alpha) \in A\}$
**compare** $(S, T)$ : boolean
    **return** $S.A \subseteq T.A \wedge S.R \subseteq T.R$
**merge** $(S, T)$ : payload
    **let** $U.A = S.A \cup T.A$
    **let** $U.R = S.R \cup T.R$
    **return** $U$
```

for each approach. State-based objects have the advantage of being simpler to reason about, given that all the information is captured by their state. Moreover, because this state is transmitted as part of the *merge* procedure between two replicas, updates may be lost along the way, applied multiple times, or in different orders. This also constitutes a disadvantage since sending whole states is an expensive network operation for large objects. On the other hand, op-based objects are more complex to specify since we need to reason about history and to deliver the updates in a causal-consistent manner according to *happens-before*. This puts pressure on the communication channel which has to be reliable and has to guarantee the same order of message delivery to all replicas. The advantage comes in the form of a greater expressive power for type specification, e.g. there is no *merge* method which has to compute a LUB, and smaller payloads.

Examples of current CRDT implementations [11] include: a) replicated counters: G-Counter, PN-Counter, Non-negative, b) registers, or mutable shared variables: LWW-Register, MV-Register, c) sets: G-Set, 2P-Set, U-set, PN-Set, OR-Set, d) graphs: 2P2P-Graph, Add-only monotonic DAG.

III. Existing Replicated Set Designs

The set data type is a pervasive abstraction, used either directly or as a building block in more complex types, such as maps or graphs. The supported update operations are *add* and *remove* which add and, respectively, remove an element to and from the set. These operations do not commute and, therefore, a CRDT set will be only an approximation to the sequential specification of a set. Many designs for replicated sets have been proposed [11] and they mainly differ in which operation takes precedence in a *add*(e) $\|$ *remove*(e) situation. In this section we give examples of some specifications and argue that the OR-Set does not present any anomaly as the others do, which could lead to a counter-intuitive behavior.

G-Set. The most basic CRDT implementation of a set, *Grow-Only Set*, allows for *add* and *lookup* operations. Both state-based and op-based versions have a set as payload. To prove that this is a CmRDT it is easy to see that *add*(e) is commutative, being based on a set-union operation between the payload and $\{e\}$. In the state-based approach, as shown in Specification 1, the partial order on states S and T is given by $S \sqsubseteq T \iff S \subseteq T$. Then, the *merge* operation defined as *merge*$(S, T) = S \cup T$ computes the LUB in the monotonic semilattice $(S, \sqsubseteq)$. And so, G-Set is also a CvRDT[2].

2P-Set. A *Two-Phase Set* brings the option to remove an element. However, once an element has been removed, it cannot be added again to the set. The principle is to use two G-Sets, one for adding and another for removing (also known as *tombstone set*). Removing an element is conditioned by being present in the set at source. The state-based variant is shown in Specification 2. The payload consists of set A for

[2]For proofs on the rest of the constructs, the reader is referred to [11].

adding and set R for removing. Adding or removing the same element twice or adding an already removed element has no effect.

U-Set. If we guarantee that each element in the set is unique and that an *add*(e) is delivered before *remove*(e), the tombstone set becomes redundant and can be discarded because the causal delivery criteria is met. This new data structure is called *U-Set*.

PN-Set. An alternative solution is to associate with each element a CRDT counter (initially set to 0) which is increased when the element is added to the set and decreased when it is removed. If the counter is negative, it means that the element is not in the set, and *add* operation will not have any effect. Thus a PN-Set has the anomaly that after adding a previously removed element to an empty set, it remains empty. This may not always be the intended semantics, despite the fact that PN-Set converges: it combines two CRDTs, a set and a counter.

OR-Set. The previously described set structures, although practical, have counter-intuitive behaviors. For example, the 2P-Set does not allow adding an element after it has been removed, while the PN-Set has the problem showed above. The *Observed-Removed Set*, introduced in [11], is closer to the usual set semantics. The new approach is to uniquely tag each added element. When removing an element, only associated tags observed at the source are removed.

Specification 3 describes the usual supported operations for the op-based variant. The payload is a set of pairs (e, tag). Method *add*(e) generates a new unique tag at source in the prepare-update phase and then sends it to all replicas which insert it into their payload in the effect-update phase. In this way, two additions of the same element are distinguished by their tags, but *lookup* masks the duplicates. Method *remove*(e) gathers all tags associated with e at source and sends them to all replicas which remove the corresponding pairs from their local payloads. Because a *remove*(e) will only remove locally observed elements, a concurrent *add*(e) $\|$ *remove*(e) will give precedence to *add*(e), in contrast to the 2P-Set.

The state-based approach is presented in Specification 4. Here the payload contains two sets, A for added elements and R for removed elements. When adding an element e, like in the op-based approach, a new unique tag α is generated and the pair (e, α) is inserted into the A set. The *remove* operation again generates a unique tag β, associates it with all matching pairs from A, and stores the result in the R set. To test if an element is in the OR-Set, we just need to verify if it is in A and not in R.

IV. A Delta-based Synchronization Algorithm

As seen in Section III, there are different ways of constructing a CRDT set data structure. OR-Sets are very intuitive and do not suffer from the semantics anomalies encountered in the other set specifications. Our goal is therefore to have the robustness of state-based OR-Set corroborated with the transfer efficiency of the op-based one. This section introduces our first contribution: improvement to the state-based OR-Set specification to transfer only deltas between replicas instead of full states, called *delta-based synchronization* (*merging*) algorithm[3], in Specification 5.

Specification 5 OR-Set with delta-based synchronization

```
payload $A = \varnothing, R = \varnothing, T = [\,]$
query $lookup(e)$ : boolean
    return $\exists(e, t, r) \in A \wedge \nexists(e, t, r, t', r') \in R$
update $add(e)$
    let $r = replica()$
    let $t = T[r] + 1$
    $A := A \cup \{(e, t, r)\}$
    $T[r] := t$
update $remove(e)$
    pre $lookup(e)$
    let $r' = replica()$
    let $t' = T[r'] + 1$
    $R := R \cup \{(e, t, r, t', r') \mid \exists(e, t, r) \in A\}$
    $T[r'] := t'$
compare $(S_1, S_2)$ : boolean
    return $S_1.A \subseteq S_2.A \wedge S_1.R \subseteq S_2.R \wedge S_1.T[i] \leq S_2.T[i], \forall i$
merge $(S_1, S_2)$ : payload
    let $A' = \{(e, t, r) \in S_2.A \mid S_1.T[r] < t\}$
    let $R' = \{(e, t, r, t', r') \in S_2.R \mid S_1.T[r'] < t'\}$
    let $P.A = S_1.A \cup A'$
    let $P.R = S_1.R \cup R'$
    let $P.T = max(S_1.T, S_2.T)$
    return $P$
```

In addition to the original OR-Set, the payload now has also a *timestamp vector* T which has as many components as there are replicas and for which $T[r]$ records the latest known version of replica r. For this purpose, it is assumed that each replica has a unique identifier that can be retrieved through the function *replica* and that T can be indexed with this identifier. Adding a new element e at replica r increments the corresponding component $T[r]$ to obtain t and inserts the tuple (e, t, r) into set A. Compared to the basic OR-Set, the change was essentially to split the tag which uniquely identified each element into the pair (t, r). In this way, the elements still remain tagged, but now we also have the information about the partial order of updates occurring at each replica, i.e. we know that tuple (e, t, r) was added before tuple (e', t', r) at replica r if $t < t'$. Removing an element uses the same principle. Being an OR-Set data type, only locally observed elements at the source are removed. The logical clock corresponding to the replica is increased again to keep track of this update. Looking up an element e in the set translates to verifying if there is an added tuple containing e and does not exist a corresponding remove tuple. In order to merge, we first send T to the other side, compute here the missing updates in A' and R', i.e. tuples whose timestamps are greater than the logical clock, send them back together with the remote T, and finally append the updates and update the local timestamps.

Therefore, T acts as a version vector [12], which guarantees the partial order between updates. Also, due to the transitivity property of the version vectors, each *merge*(S_1, S_2) includes

[3]Synchronization here has the meaning of updates propagation between replicas, and not that of a consensus required in the case of strong consistency model.

not only the updates originated at S_2 but also those from S_3 which were pulled by S_2 but not by S_1. One limitation of this approach is that indexing T requires a static mapping from a global replica identifier to an integer. Dynamic version vector maintenance using interval tree clocks [13] may alleviate this problem however.

Delta-based synchronization maintains the CRDT properties: We consider the partial order $(S, \sqsubseteq)$, where $\sqsubseteq$ is given by the *compare* method in the specification. Both *add* and *remove* methods add elements to the payload and increment T and therefore advance the state in the partial order. Furthermore, for any two sets X and Y, it is known that the following holds: $X \cup Y = X \cup (Y \setminus X)$. So *merge* basically computes the union of the added and, respectively, removed sets using the right-hand side formula and the maximum of the two timestamp vectors. Hence we have *merge*$(S_1, S_2) = S_1 \sqcup S_2$ (LUB). $\square$

V. Sharding

The next improvement to the OR-Set is partitioning, or sharding, a replica into many disjunctive subsets which can be stored individually on different machines. Each replicated set can reside in a cluster, as illustrated in Fig. 1. Here Replica 1 is sharded in 3 subsets, Replica 2 is sharded in 2 subsets, while Replica 3 is stored entirely on one machine. This OR-Set data type where each replica set is sharded into subsets will be referred to as **Sharded OR-Set** (**SOR-Set**).

In order to coordinate incoming requests for each of the replicated set, a client entity should be used to forward the *add*(e), *remove*(e), and *lookup*(e) operations to the corresponding subset. For this purpose, the client can employ any partitioning function, such as a hash function with uniform distribution, which maps each element e to a shard. The client is also responsible for initiating the *merge* operation between two clusters to pull the updates from all shards in the remote cluster and distribute them according to the same hash function to the shards in the local cluster.

Specification 6 synthesizes the usual state-based operations. Each replica i of the set is stored in a *replica cluster* rc_i. Inside the cluster rc_i, the set is partitioned into $|rc_i|$ subsets, called *replica shards* rs_i^j. Therefore, any shard is uniquely identified by the pair of identifiers (rc, rs). Based on this observation, instead of using a timestamp vector to keep track of the latest versions for the replicas as in the previous section, a vector of timestamp vectors T is used. T has as many components as there are clusters, while $T[rc_i]$ has $|rc_i|$ components, one for each shard. Like before, each cell $T[rc][rs]$ stores the latest version of the logical clock of the shard (rc, rs). Since when

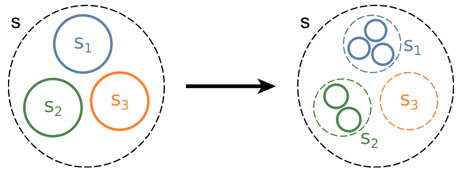

Fig. 1: Sharding of OR-Sets

Specification 6 SOR-Set with delta-based synchronization

```
payload $A_i^j = \varnothing, R_i^j = \varnothing, T_i^j = [\,][\,], \forall j \in \{1, \ldots, |rc_i|\}$
            $\triangleright$ $rc_i$ - Replica cluster $i$; $rs_i^j$ - Replica shard $j$ of $rc_i$
query $lookup_i(e)$ : boolean
    let $j = hash_i(e)$
    return $\exists(e, t, rc, rs) \in A_i^j \wedge \nexists(e, t, rc, rs, t', rc', rs') \in R_i^j$
update $add_i(e)$
    let $j = hash_i(e)$
    let $rc = rc_i$
    let $rs = rs_i^j$
    let $t = T_i^j[rc][rs] + 1$
    $A_i^j := A_i^j \cup \{(e, t, rc, rs)\}$
    $T_i^j[rc][rs] := t$
update $remove_i(e)$
    pre $lookup_i(e)$
    let $j = hash_i(e)$
    let $rc' = rc_i$
    let $rs' = rs_i^j$
    let $t' = T_i^j[rc'][rs'] + 1$
    $R_i^j := R_i^j \cup \{(e, t, rc, rs, t', rc', rs') \mid \exists(e, t, rc, rs) \in A_i^j\}$
    $T_i^j[rc'][rs'] := t'$
compare $(rc_x, rc_y)$ : boolean
    let $\tilde{T}_x = version(rc_x)$
    let $\tilde{T}_y = version(rc_y)$
    return $(\bigcup_j A_x^j \subseteq \bigcup_k A_y^k) \wedge (\bigcup_j R_x^j \subseteq \bigcup_k R_y^k) \wedge (\tilde{T}_x \leq \tilde{T}_y)$
                        $\forall j \in \{1, \ldots, |rc_x|\}; \forall k \in \{1, \ldots, |rc_y|\}$
merge $(rc_x, rc_y)$ : payload
    let $\tilde{T}_x = version(rc_x)$
    let $\tilde{T}_y = version(rc_y)$
    $\forall j \in \{1, \ldots, |rc_y|\}$
        let $A' = \{(e, t, rc, rs) \in A_y^j \mid \tilde{T}_x[rc][rs] < t\}$
        let $R' = \{(e, t, rc, rs, t', rc', rs') \in R_y^j$
                $\mid \tilde{T}_x[rc'][rs'] < t'\}$
    $\forall j \in \{1, \ldots, |rc_x|\}$
        let $Z.A_x^j = A_x^j \cup \{(e, t, rc, rs) \in A' \mid j = hash_x(e)\}$
        let $Z.R_x^j = R_x^j \cup \{(e, t, rc, rs, t', rc', rs') \in R'$
                $\mid j = hash_x(e)\}$
        let $Z.T_x^j = max(T_x^j, \tilde{T}_y)$
    return $Z$
```

adding or removing an element from the source replica, it will be added or, respectively, removed from only one shard (rc, rs), i.e. the one computed by the hash function, each update can be uniquely tagged with the tuple (t, rc, rs), where t is the timestamp generated at (rc, rs).

Returning to the specification, the payload is also distributed: A_i^j, R_i^j, and T_i^j are, respectively, the set of added and removed elements and the vector of timestamps for shard rs_i^j. The set operations follow the same principle as before. Merging the state of a remote replica from cluster rc_y into the local state in cluster rc_x is also similar. We just need to compute a minimum version $\tilde{T}_x$ first for the whole cluster by combining the information from all T_x^j using $version(rc_x)$:

$$\tilde{T}_x[rc][rs] = \begin{cases} max(\bigcup_{j \in \{1,\ldots,|rc_x|\}} T_x^j[rc][rs]) & \text{if } rc = rc_x, \\ min(\bigcup_{j \in \{1,\ldots,|rc_x|\}} T_x^j[rc][rs]) & \text{otherwise} \end{cases}$$

$$\forall rc = rc_i, i \in \{1, \ldots, N\}; \forall rs = rs_i^j, j \in \{1, \ldots, |rc_i|\}.$$

We set the minimum from all T_x^j component-wise, except for $\tilde{T}_x[rc_x]$, where we choose the maximum instead since each shard in rc_x increments its own counter only. We do the same for cluster rc_y.

Some important observations are worth mentioning. First, the *merge* operation remains unobtrusive like for all CRDTs: clients can issue requests to the set while the operation progresses in the background. Since the minimum version $\tilde{T}_y$ is computed first, the remote cluster rc_y can meanwhile process any subsequent updates. They will be pulled with the next merge. Analogously, because at the end each T_x^j is updated to the maximum between the current one and the remote one component-wise, the local cluster can in this time process any incoming client requests. Therefore, both sets can be updated while the synchronization takes place.

Second, this algorithm is resilient to shard failures in both local and remote clusters. An unreachable shard in the local cluster leads to a potential bigger $\tilde{T}_x$ except for $\tilde{T}_x[rc_x]$. This means that not all updates will be fetched. As soon as the failed shard restores, its lagging timestamp will lead to a smaller $\tilde{T}_x$ and the next merge will thus include the missing updates plus some of the already fetched ones. For the remote cluster, an unreachable shard has the same consequence: $T_x^j := max(T_x^j, \tilde{T}_y)$ will set smaller values in $T_x^j[rc_y]$ and missed updates will be fetched with the next merge after the shard restores. Optionally, a master-slave replication scheme can be used to ensure fault-tolerance for any shard.

Sharding maintains the CRDT properties: Consider a replica state as $s_i = (A_i, R_i, \tilde{T}_i)$, where $A_i = \bigcup_{j\in\{1,\dots,|rc_i|\}} A_i^j$ and $R_i = \bigcup_{j\in\{1,\dots,|rc_i|\}} R_i^j$. Thus, a set is characterized by the contributions of all its subsets. The partial order is then $(S, \sqsubseteq)$, $\forall s_i \in S$ and $\sqsubseteq$ given by *compare* method. Update operations *add* and *remove* advance the state in the partial order as they both add elements to the set and increase $\tilde{T}$. *Merge* computes the set union between A_x and A_y and between R_x and R_y, respectively. Also, because each T_x^j is updated with the maximum between T_x^j and $\tilde{T}_y$, the newly obtained $\tilde{T}_x$ will be the maximum between $\tilde{T}_x$ and $\tilde{T}_y$. Therefore *merge* computes the LUB. □

VI. Garbage Collection

As seen in Specification 6, update procedures add tuples to either A or R sets, which lead to an increase of database in size. If we want to always have a complete history for a replica, then the behavior may conform to these requirements. However, due to space constraints, this assumption is usually not practical. Hence, this section introduces an automatic garbage collection mechanism for removing, or *expiring*, tuples from these sets after a specified time interval. To this extent, elements are considered to have limited lifetime in the store, setting that can be decided on a case-by-case basis. For example, if the set tracks statistics about IP addresses used for logging into a user account, we may not be interested in IPs older than one month.

Any $(e, t, rc, rs) \in A$ in the SOR-Set will be referred to as an *ADD(e)* tuple and any $(e, t, rc, rs, t', rc', rs') \in R$ as an *RMV(e)* tuple. These tuples are generated either when the client calls *add* and *remove* methods, or through the synchronization process. Lookup semantics states that *lookup*(e) should return *true* if e is in the SOR-Set and *false* otherwise. The following theorem on tuples expiration can now be formulated.

Theorem (Tuples expiration). *If, at any given shard, the tuples corresponding to any element e, ADD(e) and RMV(e), are expired in the same order in which they were originally inserted, then the lookup semantics are preserved.*

Proof: Let us first consider update operations occurring at one replica with no synchronization taking place. There are two cases: i) $ADD(e) \rightarrow RMV(e)$, meaning *ADD(e)* is inserted before *RMV(e)*. The expected return value for *lookup*(e) after these operations are executed is evidently *false*. If *ADD(e)* expires first and *RMV(e)* expires later, then the semantics does not change. If, however, expiration occurs in reverse order, there will be a time window when *ADD(e)* is present, but *RMV(e)* not. In this interval, a *lookup*(e) call will return *true*, which will change the expected semantics. ii) $RMV(e) \rightarrow ADD(e)$. The proof follows the same rationale.

Consider now the situation when tuples propagate from one shard to another. Again there are two cases: i) $ADD(e) \rightsquigarrow RMV(e)$, which symbolizes that *ADD(e)* was originally inserted at one shard, fetched through replica synchronization and then a *RMV(e)* was inserted locally. As soon as the remote *ADD(e)* is inserted in the local set, this case reduces to the corresponding sequential one from before: $ADD(e) \rightarrow RMV(e)$ and both tuples should be expired in the same order in which were originally inserted. If the *RMV(e)* is inserted before *ADD(e)* reaches the local shard, these updates are concurrent and $ADD(e)$ wins: *lookup*(e) will return *true* as long as the $ADD(e)$ is not expired, which is what we expect. ii) $RMV(e) \rightsquigarrow ADD(e)$. Similarly, the case reduces to the sequential $RMV(e) \rightarrow ADD(e)$. □

The following changes could be made to the SOR-Set specification in order to include an automatic, asynchronous garbage collection while maintaining the lookup semantics. We associate a *time-to-live* (TTL) value with each tuple when it is inserted into the corresponding set through an *add* or *remove*. This value represents the time interval from the moment it was inserted after which the tuple will expire. In this way, tuples older than a specified period are considered to be no longer relevant and can be safely discarded. Data stores such as Redis [14] or Cassandra [15] offer support for setting TTL attributes to records and automatic removal for expired ones. Otherwise, a simple periodic scan-and-remove process on the database can be used. To ensure that tuples

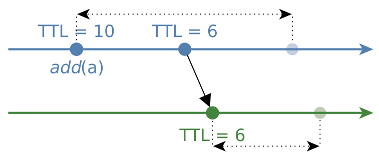

Fig. 2: Preserving TTL on updates propagation

corresponding to e expire in the order in which they were added, first, it is sufficient to stamp them with the same value $TTL(e)$. Tuples corresponding to different values may be stamped with different TTLs. Second, as shown in Fig. 2, when copying the tuples to other shards, their remaining TTL should be preserved, i.e. the current TTL at the remote replica is transferred together with the tuple to the local replica. By doing this, tuples will expire at the local replica in the same order as they do at the remote one.

We note that it is not a requirement for having the same physical clock speed on all machines or for having their clocks periodically synchronized. What is needed is only a partial order on the tuples expiration as stated by the above theorem. Preserving the TTLs for tuples when propagating them across different shards evidently does not imply that there is a global time point when all copies of one tuple are expired simultaneously. In fact, copies of the tuples in local cluster will expire shortly after the original ones in remote cluster have expired. However, this does not invalidate the lookup semantics according to the tuples expiration theorem.

Garbage collection maintains the CRDT properties: We consider first the case when no sharding is used. A new partial order can be defined by the relation $S_1 \sqsubseteq S_2 \iff S_1 \subseteq S_2 \vee S_1 \equiv (S_1 \cap S_2)$. The first term holds when no tuples are expired and thus either *add* or *remove* operation increases the corresponding set like before. If by the time we apply any operation, some tuples are expired from S_1, then the states containing old non-expired tuples from before and after the update are considered equivalent, i.e. any *lookup*(e) method on either S_1 or $S_1 \cap S_2$ returns the same result. It is easy to see that, relative to $\sqsubseteq$, the updates always advance the states in the partial order. Taking sharding into account, we can simply consider the union of all A and, respectively, R sets in one cluster as in Specification 6: $S_i = \bigcup_{\forall j \in \{1,\ldots,|rc_i|\}} S_i^j$, where S is A or R. From this point, the proof follows the same rationale as for the SOR-Set. $\square$

VII. Library Implementation

To test these concepts, we implemented a Java client library for the SOR-Set which can connect to any replica cluster and provide access to the usual set operations: adding, removing, looking up an element, or synchronizing with other replica clusters.

For the database server we used Redis [14], a widely used, open-source, in-memory, key-value store. Redis data model is a dictionary that maps keys to values. The keys can be only strings, while values can be strings, lists, sets, sorted sets, or hashes. Important Redis features include persistence, replication, transactions, pipelining, and Lua scripting. In addition, it has support for associating timeouts with keys: after the timeout has expired, the key is automatically deleted. Our client communicates with Redis via the Jedis library [16].

The payload for each shard is stored in a separate Redis database, or *store*, using the schema from Listing 1, where underlined words represent hard-coded strings and non-underlined ones are to be replaced with their corresponding values.

Listing 1 Redis database schema for a SOR-Set shard

```
1: timestamp:$rc$:$rs$ $\rightarrow t$                              ▷ Integer string
2: element:$rc.rs.id$  $\rightarrow$ value $\rightarrow e$                    ▷ Hash
                     $\rightarrow$ add.t $\rightarrow t$
                     $\rightarrow$ add.rc $\rightarrow rc$
                     $\rightarrow$ add.rs $\rightarrow rs$
                     $\rightarrow$ rmv.t $\rightarrow t'$
                     $\rightarrow$ rmv.rc $\rightarrow rc'$
                     $\rightarrow$ rmv.rs $\rightarrow rs'$
3: index:$rc$:$rs$ $\rightarrow [t$:$rc.rs.id]$                         ▷ List
4: ids:$e \rightarrow rc.rs.id$                                  ▷ Set
5: element:next.id $\rightarrow id$                             ▷ Integer string
```

Each cell of the vector of timestamps, $T[rc][rs]$, is stored at key timestamp:rc:rs. Instead of using different sets for added and removed tuples, we combine and store them together as *elements*. An element contains: i) the string value e, ii) information about a adding: timestamp t and ids for the source replica cluster and shard, (rc, rs), iii) similar information for removing: t', rc', rs'. Each element is stored in the hash at key element:$rc.rs.id$, where $rc.rs.id$ represents a global unique id: rc and rs are the ids which uniquely identifies the source shard and id is a per-shard counter stored at element:next.id key which is incremented with each new element insertion.

In Specification 6 of the SOR-Set, the *merge* method filters all tuples added or removed after a given timestamp. For this purpose, an index is kept as a list of element ids sorted by their timestamp. With each add or remove of an element at shard (rc, rs), its id is appended to the list index:rc:rs. Since the index is kept per shard and timestamps at each shard are monotonically increasing (adding and removing always increases the local timestamp), index:rc:rs is guaranteed to be always sorted. Thus, filtering new elements is very efficient. Adding the same value e multiple times to the set creates a new element for each operation. A second index stored at ids:e keeps all the element ids corresponding to value e, needed for *remove*(e) and *lookup*(e) methods.

Add. Listing 2 gives the pseudocode for *add* operation. First, the logical clock of the shard and the local id counter are incremented. `incr` is an internal Redis command which increments the number stored at the specified key by one[4]. Next, a hash is created to store the new element and its expiration time is set. Last two lines update the two indices previously discussed.

Remove. The *remove* method is described in Listing 3. Again, removing an element e from an SOR-Set consists in getting all $ADD(e)$ tuples and tagging them as removed. Thus, on line 3 all element ids for value e are retrieved using index ids:e. For each element stored at element:gid, if it is not yet expired, i.e. the key still exists in the database, the corresponding fields are populated. Finally, the procedure updates the expiration period of the element and pushes the

[4]The rest of Redis commands will not be described as their usage will be easily deduced from the context.

Listing 2 Redis SOR-Set: *add*

```
procedure ADD($e$, $rc$, $rs$, $ttl$)
    $t \leftarrow$ incr $\underline{\text{timestamp}}$:$rc$:$rs$
    $id \leftarrow$ incr $\underline{\text{element:next.id}}$
    hmset $\underline{\text{element}}$:$rc.rs.id$  $\underline{\text{value}}$ $e$
                                $\underline{\text{add.t}}$ $t$
                                $\underline{\text{add.rc}}$ $rc$
                                $\underline{\text{add.rs}}$ $rs$
    expire $\underline{\text{element}}$:$rc.rs.id$ $ttl$
    lpush $\underline{\text{index}}$:$rc$:$rs$ $t$:$rc.rs.id$
    sadd $\underline{\text{ids}}$:$e$ $rc.rs.id$
```

Listing 3 Redis SOR-Set: *remove*

```
procedure REMOVE($e$, $rc'$, $rs'$, $ttl$)
    $t' \leftarrow$ incr $\underline{\text{timestamp}}$:$rc'$:$rs'$
    $ids \leftarrow$ smembers $\underline{\text{ids}}$:$e$
    for all $gid$ in $ids$ do
        if exists $\underline{\text{element}}$:$gid$ then
            hmset $\underline{\text{element}}$:$gid$  $\underline{\text{rmv.t}}$ $t'$
                                   $\underline{\text{rmv.rc}}$ $rc'$
                                   $\underline{\text{rmv.rs}}$ $rs'$
            expire $\underline{\text{element}}$:$gid$ $ttl$
            lpush $\underline{\text{index}}$:$rc'$:$rs'$ $t'$:$gid$
```

new timestamp together with the id to the index list. After removal, the new timestamp t' will be ahead of the old one, t, in this list, which the expected behavior: the remove happened after the add.

Lookup. The *lookup* method searches for the existence of at least one $ADD(e)$ tuple for which there is no corresponding $RMV(e)$. If there is such tuple, then element e is in the set. This is exactly what Listing 4 does. First, ids for all elements e are retrieved. Next, for each element $(e, t_1, rc_1, rs_1, t'_1, rc'_1, rs'_1)$, if it is an $ADD(e)$ tuple ($t'_1 = null$, its removed timestamp is not set) and does not exist any corresponding $RMV(e)$ tuple $(e, t_2, rc_2, rs_2, t'_2, rc'_2, rs'_2)$, such that $(t_1, rc_1, rs_1) = (t_2, rc_2, rs_2)$ and $t'_2 \neq null$, then $true$ is returned.

Procedures ADD, REMOVE, and LOOKUP execute atomically and in isolation with other Redis commands on the store.

Merge. The code for last method is given in Listing 5. For lack of space we present only the main subroutines. On lines 2 and 3 we compute the minimum versions for both the local and the remote clusters using the formula explained in Section V. Based on the local version $\tilde{T}_x$, GETUPDATES fetches the updates from remote cluster rc_y. This is done in 2 steps: first it gets the ids of the updates using $\underline{\text{index}}$:rc:rs and then it retrieves the actual elements. ADDUPDATES distributes these elements to the stores in the local cluster according to $hash_x$ function. Adding an update element to the store is an operation similar to *add*, except that the logical clock is not incremented and the TTLs are the ones retrieved before. Setting the logical clocks is done at the end in the UPDATETIMESTAMPS subroutine according to the formula $T_x^j := max(T_x^j, \tilde{T}_y), \forall j \in \{1, \ldots, |rc_x|\}$.

Listing 4 Redis SOR-Set: *lookup*

```
function LOOKUP($e$)
    $ids \leftarrow$ smembers $\underline{\text{ids}}$:$e$
    for all $gid_1$ in $ids$ do
        if exists $\underline{\text{element}}$:$gid_1$ then
            ($t_1$, $rc_1$, $rs_1$, $t'_1$) $\leftarrow$
                hmget $\underline{\text{element}}$:$gid_1$ $\underline{\text{add.t}}$ $\underline{\text{add.rc}}$ $\underline{\text{add.rs}}$ $\underline{\text{rmv.t}}$
            if $t'_1 = $ null then
                $lookup \leftarrow$ true
                for all $gid_2$ in $ids$ do
                    if exists $\underline{\text{element}}$:$gid_2$ then
                        (_, $t_2$, $rc_2$, $rs_2$, $t'_2$, $rc'_2$, $rs'_2$) $\leftarrow$
                            hgetall $\underline{\text{element}}$:$gid_2$
                        if ($t_1$, $rc_1$, $rs_1$) = ($t_2$, $rc_2$, $rs_2$) and
                            $t'_2 \neq$ null then
                            $lookup \leftarrow$ false
                            break
                if $lookup = $ true then
                    return true
    return false
```

Listing 5 Redis SOR-Set: *merge*

```
1: procedure MERGE($rc_x$, $rc_y$)
2:     $\tilde{T}_x \leftarrow$ VERSION($rc_x$)
3:     $\tilde{T}_y \leftarrow$ VERSION($rc_y$)
4:     $updates \leftarrow$ GETUPDATES($rc_y$, $\tilde{T}_x$)
5:     ADDUPDATES($rc_x$, $hash_x$, $updates$)
6:     UPDATETIMESTAMPS($rc_x$, $\tilde{T}_y$)
```

VIII. EVALUATION

This section presents the results obtained for evaluating the SOR-Set client library. The test systems are equipped with Intel Xeon E5520 dual quad core CPUs with HyperThreading support running at 2.27GHz and with 24GB of RAM, interconnected through 1Gbps network interfaces. For the datastore, Redis version 2.6.0-rc6 is used.

The purpose of the first benchmark is to measure how the average time needed to merge two replicated sets changes as the database size increases. Test configuration includes 16 Redis instances running on one machine representing Replica A, each instance storing one shard of the set. For Replica B another machine with identical configuration is used. The client library is deployed on a third machine. The methodology for measuring is: add 1 million 32-byte uniform randomly generated elements to Replica A, measure the time for merging into Replica B using a pool of 16 threads, and then repeat the process.

Results are presented in Fig. 3. Here are also included the average timings for each subroutine of the *merge* procedure described in Listing 5 relevant to these measurements: getting the element ids, fetching the actual elements from the remote cluster, and adding the elements to the local cluster. The first observation is that delta-based synchronization algorithm scales well with the database size, showing a synchronization cost which is constant per update and per million set elements. Since the number of updates between each merge operation is constant, the timings are also relatively constant. Thus, the

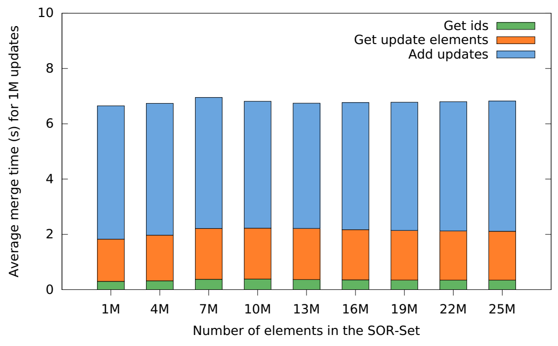

Fig. 3: Delta-based synchronization

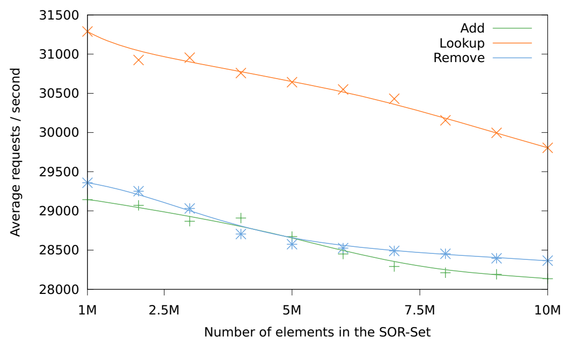

Fig. 4: Throughput for set operations

merge procedure has a time complexity proportional to the number of updates, i.e. delta size, and not to the database size. Second, from this plot the average throughput for merging can be computed to 125,000 update elements per second.

The second benchmark measures the average throughput of all the basic set operations. For this purpose, a machine with 16 Redis servers acting as a replica cluster is used. The client library is deployed on another machine to perform the test: add 1 million 32-byte uniform randomly generated elements to the set, look them up, remove them, and then repeat the process with more elements. The drop in throughput in Fig. 4 can have one of two causes: either the operations have time complexity proportional to the database size, or Redis incurs performance penalty as its database increases. Listings 2, 3, and 4 show that only $\mathcal{O}(1)$ Redis operations are used, assuming that same values are not inserted in the set. This a reasonable assumption since 10 million elements are generated, each chosen with the same probability from a $2^{8\times32}$ space, and thus leading to a low chance of collision. Therefore, updating the indices is on average a constant operation: $\underline{\text{ids}}$:e contains only one id and `lpush` $\underline{\text{index}}$:rc:rs is constant. The decline in performance may be attributed to Redis' management of its internal structures, such as the global hash table which stores all the keys. As the database size increases, Redis has to adjust the capacity of this hash, making a simple `get` operation on any key costly. This is not visible in Fig. 3 because the timings are dominated there by client operations.

The reason why a better throughput is obtained for *merge* has two causes. First, each of *add*, *lookup*, and *remove* is implemented using Lua scripts for which Redis guarantees to execute in an atomic way. This is needed to ensure that updating the elements and the indices in the database does not interleave with other Redis commands. Second, fetching the elements and distributing the updates in the *merge* procedure are done using pipelines: sending multiple Redis commands without waiting for a reply from the server, thus saving the round-trip-time of each request. Unfortunately, the same technique cannot be used for the other procedures because both *add* and *remove* increment a counter to generate the id for each element, while *lookup* must first fetch all ids of one element. This means we have to wait for a reply from Redis before calling the subsequent commands, i.e. basic set operations contain synchronous calls to Redis which make them unsuitable for pipelining. This is not considered to be a problem since these procedures are independent and are usually issued by different clients, as opposed to the subroutines of one *merge* call.

IX. Comparison with Previous Work

The fundamental principles on database replication are laid out in [17] and a number of techniques are discussed there to achieve consistency. The traditional *strong consistency* approach imposes a global total order on updates to serialize them [18]. This conflicts with availability and partition-tolerance [1] and leads to performance and scalability bottlenecks. *Sequential consistency* is another model, weaker than strong consistency, but undecidable in practice [19]. A survey on other models is presented in [20].

Techniques for achieving eventual consistency for large-scale distributed systems have been an active focus point in recent research. This is mostly due to the explosion of Internet-based and peer-to-peer services. However, the origins of the principles behind CRDTs can be found in the apparent unrelated area of file systems. The state-based approach was introduced for register-like objects, where the only operation is assignment. It is widely used in NFS [21] and AFS [22] file systems and in key-value stores such as Amazon's Dynamo [5]. The mathematical foundations were laid by Baquero and Moura [23] and later extended by Shapiro and Preguiça in their work on Treedoc [7] in order to support the operation-based approach, thus coining the term of CRDT. Examples of implementations for this second approach are found in Bayou's anti-entropy protocol [9] and the IceCube cooperative system [24]. Later, a formal definition and rigorous system model for CRDT were published in [11] and [6]. These are the first works to engage a comprehensive and systematic study on CRDTs.

Several designs of replicated sets have been proposed, but many of them present anomalies. Amazon's Dynamo shopping cart [5] uses registers in its implementation. It takes the union of concurrent assignments, multiple values are later reduced to a single one. The problem is that a removed element can reappear. In a 2P-Set [25], adding an element after it has been removed has not effect. Furthermore, this design imposes synchronization for reclaiming the tombstones. In Section III we gave more examples for replicated sets and concluded that the OR-Set behaves intuitively.

X. Conclusions

Achieving consistency in large-scale distributed systems is not an easy task. To make things more difficult, designers need to also ensure high-throughput, low-latencies accesses to the databases. However, building reliable distributed systems demands trade-offs between consistency and availability as stated by the CAP theorem [1]. Eventual consistency is a technique of compromise, widely adopted, but lacking a rigorous theoretical foundation which makes current approaches ad-hoc and error-prone [5].

The concept of CRDTs defines replicated data types that have mathematical properties conferring them a form of eventual consistency, strong eventual consistency. This model can be described from two equivalent perspectives: a) state-based: object replicas apply updates locally and later exchange and merge their states, and b) operation-based: update operations are distributed among replicas over a reliable broadcast communication channel. Both approaches guarantee convergence towards a common state without application-level conflict resolutions, roll-backs, or consensus among replicas [4]. Because this model imposes strong constraints on the type specification, it is not universal though.

In this paper we focused on designs for conflict-free replicated sets and gave number of practical examples. We introduced two improvements to the original OR-Set specification: an algorithm for efficient delta-based synchronization and an extension for replica sharding. Lastly, we proposed a garbage collection mechanism to support lifetime-limited elements and to alleviate the problem of unbounded database growth. On the practical side, to the authors' knowledge, this is the first implementation of a CRDT in the sense of the system model described in this chapter. Proof-of-concept examples exists [26], [27], but they focus only on testing the specifications for CRDTs locally and in-memory, without a real database store support.

Based on this type, more complex structures can be built. Maps can be implemented as sets of registers and graphs can include two sets: one for vertices and another for edges.

References

[1] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.

[2] W. Vogels, "Eventually consistent," *ACM Queue*, vol. 6, no. 6, pp. 14–19, 2008.

[3] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 42–81, Mar. 2005.

[4] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 172–182.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.

[6] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, ser. SSS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 386–400.

[7] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 395–403.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, ser. PODC '87. New York, NY, USA: ACM, 1987, pp. 1–12.

[9] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*, ser. SOSP '97. New York, NY, USA: ACM, 1997, pp. 288–301.

[10] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," INRIA, Research Report RR-7506, Jan. 2011.

[12] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *IEEE Trans. Softw. Eng.*, vol. 9, no. 3, pp. 240–247, May 1983.

[13] P. S. Almeida, C. Baquero, and V. Fonte, "Interval tree clocks," in *Proceedings of the 12th International Conference on Principles of Distributed Systems*, ser. OPODIS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 259–274.

[14] "Redis," http://www.redis.io, [A key-value store].

[15] "Apache Cassandra," http://cassandra.apache.org, [A distributed structured key-value store].

[16] "Jedis," https://github.com/xetorthio/jedis.

[17] B. G. Lindsay, "Notes on distributed databases," IBM Research Laboratory, San Jose, NY, USA, IBM Research Report RJ2571(33471), Jul. 1979.

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[19] S. Qadeer, "Verifying sequential consistency on shared-memory multiprocessors by model checking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 8, pp. 730–741, Aug. 2003.

[20] D. Mosberger, "Memory consistency models," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 1, pp. 18–26, Jan. 1993.

[21] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation or the sun network filesystem," 1985.

[22] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 51–81, Feb. 1988.

[23] C. Baquero and F. Moura, "Specification of convergent abstract data types for autonomous mobile computing," Departamento de Informática, Universidade do Minho, Tech. Rep., Oct. 1997.

[24] N. Preguiça, M. Shapiro, and C. Matheson, "Semantics-based reconciliation for collaborative and mobile environments," in *COOPIS*, ser. Lecture notes in computer science, D. C. S. e. a. Robert Meersman, Zahir Tari, Ed., vol. 2888. Catania, Sicily, Italie: Springer, 2003, pp. 38–55.

[25] G. T. Wuu and A. J. Bernstein, "Efficient solutions to the replicated log and dictionary problems," in *Proceedings of the third annual ACM symposium on Principles of distributed computing*, ser. PODC '84. New York, NY, USA: ACM, 1984, pp. 233–242.

[26] "GitHub ericmoritz repository," https://github.com/ericmoritz/crdt.

[27] "GitHub dominictarr repository," https://github.com/dominictarr/crdt.