FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

in collaboration with

1&1 INTERNET AG, MÜNCHEN

Master's Thesis in Informatik

Design and Implementation of a Scalable Conflict-free Replicated Set Data Type

Andrei Deftu

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

in collaboration with

1&1 INTERNET AG, MÜNCHEN

Master's Thesis in Informatik

Design and Implementation of a Scalable Conflict-free Replicated Set Data Type

Design und Implementierung eines skalierbaren, konflikt-freien, replizierten Mengendatentyps

Author:	Dipl. Ing. Andrei Deftu
Supervisor:	Prof. Dr. Arndt Bode, TUM
Advisor:	Dipl. Ing. Alin Murarasu, TUM
Advisor:	M.Sc. Ing. Jan Griebsch, 1&1 Internet AG
Date:	October 15, 2012

I assure the single handed composition of this Master's Thesis only supported by declared resources.

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den October 14, 2012 Andrei Deftu

Acknowledgments

This thesis would not have been possible without the support of Eric Müller of 1&1 Internet AG. I wish to take this opportunity to express to express my sincere gratitude to my advisor, Jan Griebsch who offered invaluable assistance and guidance throughout all the work performed inside the company. In addition, I would like to thank Stefan Demharter, also from 1&1 Internet AG, for useful insights on this project. Last but not least I thank Alin Murarasu for his constant support, help and motivation during my whole study period at TUM.

Abstract

In the current context of increasingly popular distributed data stores, replication and database partitioning are fundamental approaches to achieve load balancing, scalability, and fault tolerance. However, this comes at the cost of synchronization between different replicas of some mutable shared object in order to maintain them consistent. *Conflict-free Replicated Data Types* (CRDTs) define new concepts to alleviate this problem. Provided that some simple mathematical properties hold, CRDTs ensure eventual consistency, such that replicas converge to a common state, equivalent to a correct sequential execution without foreground synchronization.

This thesis studies the applicability of this model with the focus on a particular CRDT, a *set* data type. An efficient algorithm for sending deltas of updates between replicas and a specification for partitioning a set replica into disjunctive subsets are introduced. Additionally, to cope with the problem of database increase in size, a garbage collection mechanism to reclaim obsolete elements from the set is discussed. This does not invalidate the CRDT properties. Lastly, the thesis shows design and implementation of a proof-of-concept client library for this data structure.

Contents

Chapter 1

Introduction

1.1 Motivation and Problem Definition

The increase in demand of highly available data, supported by the growing exposure of Internet services that make this data accessible, shows that there is an obvious need today to satisfy requirements that were never before a priority. 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 is a technique which ensures 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 different replicas of the same logical data consistent with each other. Ideally, we want the data stores to be always available, strongly consistent, and to operate flawlessly in the presence of network 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, at the cost of 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 and then to send the operations asynchronously to all the others. Thus, all replicas eventually apply all updates, possibly even in a different order. This weaker form of consistency, considered acceptable for some applications, has the advantage that data remains available when the network is partitioned. However, a complex background consensus algorithm for reconciling conflicting updates[1] 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.

The thesis studies a particular kind of data types which were designed specifically to solve this problem. *Conflict-free Replicated Data Types* (CRDTs) have simple mathematical properties which confer them *strong eventual consistency*, as defined in Section 2.4. Replicas of CRDTs are proved to converge in a self-stabilising manner without blocking client operations and without having to deal with complex conflict resolution or roll-backs. The drawback of this model is that it is not universal. Chapter 2 presents the trade-offs that should be considered for these types.

A strong motivation to study this concept comes from current demands to support frequent writes of runtime data to replicated, distributed stores, and to provide low latencies for reads. Because a consensus algorithm for conflicting updates would become a bottleneck, CRDTs are very attractive, as a replica may execute any operation locally. Consistency is achieved later during a background asynchronous phase in which all replicas eventually apply all updates. A second benefit is given by the composability nature of CRDTs: simple structures (counters, shared mutable variables, sets) can constitute building blocks in forming more complex ones, such as graphs.

1.2 Objectives

This thesis presents a study on principles of CRDTs together with associated mathematical properties in the context of data replication. Chapter 2 introduces the theoretical background from the perspective of two approaches, *state-based* and *operation-based*, and shows that CRDTs are a viable solution to the CAP theorem by intrinsically employing a form of eventual consistency, *Strong Eventual Consistency* (SEC). As mentioned in the previous section, the drawbacks of current alternatives which require consensus for conflict arbitration between concurrent updates are avoided.

The main goals of the thesis are to improve a particular CRDT, a *set* data type, and to study its applicability. To achieve these objectives, the following contributions are made:

- Because one variant of CRDTs transfer full states between replicas when propagating updates, an improvement to the set type is provided which transmits only deltas.

[1]A conflict is a combination of concurrent updates, which may be individually correct, but if taken together, would violate some invariant.

- 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.
- Since CRDTs usually lead to an increase in database size with each update operation, an asynchronous garbage collection process to reclaim obsolete elements from the set is discussed.
- Finally, these concepts are put into practice through the implementation and evaluation of a client library backed by Redis [6], a widely used key-value store.

1.3 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 presents the state-of-the-art on CRDT system model, two functional approaches: state-based and operation-based, and gives a number of examples of CRDTs known in the literature: counters, registers (mutable shared variables), sets, and graphs. In Chapter 3 the initial problem statement is reiterated in the light of this model and the design for an improved set type is described. Chapter 4 next presents the client library implementation and evaluation results for this data structure. Finally, Chapter 5 concludes the thesis with a summary of lessons learned and gives insights into possible future contributions.

Chapter 2

State-of-the-Art

This chapter describes the theoretical foundations, defines the concept of Strong Eventual Consistency (SEC), and presents two approaches of CRDTs: state-based, or Convergent Replicated Data Types (CvRDT), and operation-based, or Commutative Replicated Data Types (CmRDT). Formal specifications for some examples of data structures are given. The chapter concludes with other approaches to achieve consistency in the context of data replication, previous work on CRDTs, and a brief description of a widely used, scalable data store.

2.1 Objects, Operations, and Replication

The following system model was originally introduced in [7] and [8].

For our purposes we consider a system consisting of processes which can communicate in an asynchronous manner over a fully meshed network. The connection between two processes may drop and recover at any time without notification. The processes may also crash and recover with their memory intact.

Objects are mutable replicated data types. An object has:

(a) an identity;

(b) a content, called its *payload*, which may have any number of immutable primitive data types (integers, strings, sets, tuples, etc.) or other objects;

(c) an initial state;

(d) an interface which defines the *operations* supported by the object.

Two objects having the same identity, but located in different processes are called

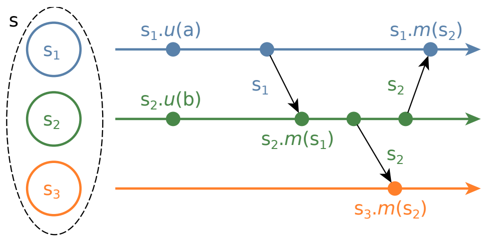


Figure 2.1: State-based replication

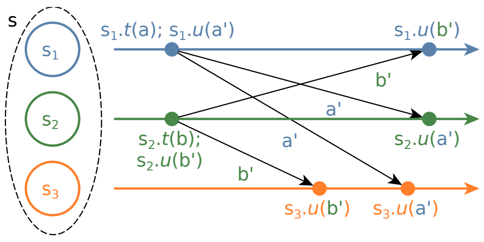


Figure 2.2: Op-based replication

replicas of each other. Without loss of generality, the replication of one object across all processes will be considered.

Clients can query and modify the state of an object by calling the operations in its interface against a replica, called the *source* replica. Queries execute only locally, i.e. entirely by the corresponding source process. Operations which modify the state, or update operations, execute in two phases. In the first phase, called *at-source*, operations are executed atomically at the source to perform some initial processing, assuming that the *source precondition* is satisfied (e.g. an element can be removed from a replicated set only if it is present in the set at the source). If the precondition is met, then the operation is said to be *enabled* and can execute as soon as it is invoked. Preconditions may be omitted if the operation can be always executed. In the second phase, called *downstream*, the update is transmitted asynchronously from source to all other replicas. With regards to this, there are two styles of replication recognized by the literature [3]: state-based and operation-based, explained below.

2.2 State-based Replication

In the *state-based* replication style, 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. In this way, every update eventually reaches every replica directly or indirectly. This process is illustrated in Figure 2.1.

An object is represented as a tuple (S, s^0, f), where S is the set of possible states, $s \in S$ is the current state, or *payload* (initial state is s^0), and f can be any of q - *query*, u - *update*, or m - *merge* operations. The notation $f_i^k(a)$ refers to the k^{th} execution of method f with arguments a at replica i. The position of operation f in the time-ordered sequence of all executions f at replica i is denoted by $K_i(f)$. Thus $K_i(f_j^k(a)) = k$ if $i = j$ and not defined otherwise. Every k^{th} update operation u at replica i, changes the state from s_i^{k-1} to s_i^k through the transition $s_i^k = s_i^{k-1} \bullet f_i^k(a)$. Queries do not change the state, i.e. if $s' = s \bullet q$, then $s' \equiv s$, meaning that states s' and s are equivalent (all queries return the same results for both).

Definition 1 (Casual history – state-based). *Causal history for an object is* *defined as the series $H = [h_1, \ldots, h_n]$, where h_i denotes the history of update operations at replica i: $h_i = [h_i^0, \ldots, h_i^k, \ldots]$. Initially, $h_i^0 = \varnothing$ for all replicas $i = 1, \ldots, n$. If the k^{th} execution of an operation at i is: i) a query q, then the causal history does not change, $h_i^k = h_i^{k-1}$; ii) an update u, then it is added to the causal history, $h_i^k = h_i^{k-1} \cup u_i^k(a)$; iii) a merge m with a remote state $s_{i'}^{k'}$, then the remote causal history is added to the local one, $h_i^k = h_i^{k-1} \cup h_{i'}^{k'}$.*

An update is said to be *delivered* at some replica if it is included in that replica's causal history. An update u *happens-before* another update u' if at the time of execution of u', u is already delivered at that replica: $u_i \rightarrow u'_i \iff u_i \in h_i^{k-1} \wedge K_i(u'_i) \geq k$. Two updates are *concurrent* if no one happens before the other: $u \parallel u' \iff u \not\rightarrow u' \wedge u' \not\rightarrow u$.

In order to have consistency among all replicas, the system has to deliver every update to every replica. Intuitively, process i sends its state s_i to process j. Here, s_i is merged into s_j by applying method m. We do this infinitely often and choose which replica to send the updates to according to any algorithm we desire. There are well known protocols in the literature that do this in a fault-tolerant manner, such as gossip or anti-entropy [9, 10], which ensure eventual distribution of all updates in the system.

2.3 Operation-based (Op-based) Replication

In the *operation-based* approach (*op-based* for short), the system transmits only update operations across replicas, and not whole states, as illustrated in Figure 2.2. In this case, the *at-source* phase has no side-effects. 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 downstream, the operation is executed immediately at source and asynchronously at all other replicas if the downstream precondition is met. This second phase cannot return results and has to execute atomically.

Because an op-based object does not have a merge operation, the update method is split into (t, u) pair, where t is the *prepare-update* method which has no side-effects and u is the *effect-update* method. We also require that u follows immediately after t, i.e. if $f_i^{k-1} = t$, then $f_i^k = u$. As explained above, t executes only at the source replica and does not modify the state, while u executes at all replicas.

Definition 2 (**Casual history – op-based**). *Similar to state-based replication, the initial history is empty at all replicas: $h_i^0 = \varnothing$. Queries do not change the causal history. After executing the update $u_i^k(a)$ at replica i in downstream, it is added to the local causal history, $h_i^k = h_i^{k-1} \cup u_i^k(a)$.*

As in the state-based case, it is required that all updates eventually reach the causal history of all replicas. This means we need a reliable broadcast communication system that delivers every update in an order $<_d$, called *delivery order*, assuming that the downstream precondition holds. *Causal delivery* $<_\rightarrow$ can be thus defined as follows: $(t, u) \rightarrow (t', u') \iff u \in h_i^{k-1}$, where t' executes at replica i, and $K_i(t') \geq k$, which means that u is delivered before u' is delivered and delivery has the same meaning as for state-based objects. Thus, while related updates are executed in the same sequential order given by happened-before, concurrent updates (i.e. not ordered by $<_d$) may be delivered in any order. There are common epidemic protocols, such as Bayou's anti-entropy [10], which provide causal ordering for updates delivery.

2.4 Strong Eventual Consistency (SEC)

This section gives a formal explanation of what it means for replicas of an object to eventually converge, or to achieve eventual consistency.

Definition 3 (**Eventual Consistency (EC)**). *An object is EC if:*

- *Safety: Replicas that have delivered the same updates, eventually reach equivalent states: if $h_i = h_j, \forall i, j$, then eventually $s_i \equiv s_j$.*
- *Liveness: If an update is delivered at replica i, then it is eventually delivered at all replicas j: if $u \in h_i$, then eventually $u \in h_j$, $\forall i, j$.*

Informally, eventual consistency means that all updates are expected to propagate through the whole system, thus keeping the replicas consistent, given a sufficiently long period of time in which no updates on the data are applied. EC implies roll back mechanisms to deal with conflicting updates and consensus to ensure that all replicas arbitrate conflicts in the same way. A stronger condition is needed to avoid this:

Definition 4 (**Strong Eventual Consistency (SEC)**). *An object is SEC if it is EC and at any time correct replicas that have delivered the same updates are in equivalent states.*

As mentioned in Section 1.1, the CAP theorem states that it is impossible for a distributed system to simultaneously achieve consistency, availability and network partition tolerance. Since for most systems nowadays availability cannot be sacrificed, consistency has to be met through other, weaker constraints. In this context, SEC provides a solution to this problem, as replicas are always available for update operations (even if partitioned), while guaranteeing eventual convergence. However, since the definition implies that all updates are immediately made persistent, it means that they must have deterministic outcome which makes this model not universal.

All CRDTs described further in this thesis are proved to achieve SEC, which gives them some attractive properties: updates can be immediately applied without taking into account network failures, no distributed consensus and synchronization for conflict resolution are required, making these data structures scalable and highly responsive.

2.5 State-based CRDT: Convergent Replicated Data Types (CvRDT)

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* [11] (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.

Definition 5 (**Convergent Replicated Data Types (CvRDT)**). *A CvRDT is a state-based object $(S, \sqsubseteq, s^0, f)$ with the following properties:*

- *Its payload takes values in the semilattice $(S, \sqsubseteq)$.*
- *Updates monotonically increase the states according to $\sqsubseteq$: $s \sqsubseteq s \bullet u$.*
- *The merge operation computes the LUB between the local and remote states: $s \bullet m(s') = s \sqcup s'$.*

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

In order for a CvRDT to eventually converge, it is required that all replicas receive all updates. Because the merge operation computes the LUB and is therefore commutative and idempotent, a reliable communication channel is not necessary. Messages may be lost, delivered out of order, or multiple time, but as long as eventually all replicas receive all updates directly or indirectly via merge operations, convergence is guaranteed.

Theorem 1. *A state-based object is SEC if it satisfies the CvRDT properties, assuming that the system transmits updated states infinitely often between all replicas.*

Proof. We consider any two replicas i and j. If $s'_i = s_i \bullet m(s_j)$ and $s'_j = s_j \bullet m(s_i)$, then, by Definition 1, s'_i and s'_j will contain the same causal history, since $h_i \cup h_j = h_j \cup h_i$. Also, because merge computes the LUB and LUB is commutative, we have $s_i \sqcup s_j = s_j \sqcup s_i$ and therefore $s'_i = s'_j$ in the semilattice, or $s'_i \equiv s'_j$. This means that both liveness and safety properties are maintained and the CvRDT converges towards the LUB of the most recent updates. □

2.6 Op-based CRDT: Commutative Replicated Data Types (CmRDT)

As noted before, in the case of op-based objects a reliable broadcast channel which delivers all updates at every replica in the delivery order $<_d$ is required. For convergence, another property is needed.

Definition 6 (**Commutativity**). *Two updates (t, u) and (t', u') commute if and only if for any reachable replica in state s, where both u and u' are enabled, u (resp. u') remains enabled in state $s \bullet u'$ (resp. $s \bullet u$) and the resulting states are equivalent: $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$.*

Definition 7 (**Commutative Replicated Data Types (CmRDT)**). *A CmRDT is an op-based object for which the following properties hold:*

- *Related updates by happened-before are delivered in the order given by $<_{\rightarrow}$.*
- *Concurrent operations commute and may be delivered in any order.*

Theorem 2. *An op-based object is SEC if it satisfies the CmRDT properties assuming a reliable communication channel.*

Proof. We consider any two replicas i and j. Under the assumption of a reliable broadcast channel, the two replicas will deliver the same operations (if no new operations are generated) and thus they will have the same causal history, $h_i = h_j$. For any two operations $u, v \in h_i$: i) if they are not causally related ($u \parallel v$), then they commute and can be delivered in any order, i.e. either $K_i(u) < K_i(v)$ or $K_i(u) > K_i(v)$; ii) if they are causally related ($u \rightarrow v$), then they are delivered in the order $u <_d v$ and are applied in the same causal order in both replicas: $K_i(u) < K_i(v)$ and $K_j(u) < K_j(v)$. In all cases an equivalent abstract state is reached: $h_i = h_j \Longrightarrow s_i \equiv s_j$. □

2.7 Relation between CvRDTs and CmRDTs

In the previous sections two approaches to eventual convergence were discussed, in the form of CvRDTs (state-based approach) and CmRDT (op-based approach), which together are simply called CRDTs. Both of them provide SEC and, interestingly enough, any data type that can be implemented as a CvRDT can also be implemented as a CmRDT and vice versa. This equivalence of the two mechanisms will be formally proved in this section, but first it is necessary to outline the differences between them and to explain which one is more suitable to be used depending on the needs.

CvRDTs 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, the requirements on the communication channel are not strong: messages may be lost along the way or received in different orders. This also constitutes a disadvantage because sending whole states is an expensive network operation for large objects. On the other hand, CmRDTs are more complex to specify since we need to reason about history and to deliver the updates in a causal-consistent manner. 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.

2.7.1 Op-based Emulation of a State-based Object

Any SEC state-based object can be emulated by a SEC op-based object having the same client interface. Obviously, queries may be ignored because they pose no problems. Regarding updates, the original u method of the CvRDT is split into the pair (t, u') of the corresponding CmRDT. For this, it can be considered that $t(a)$ and $u(a)$ accept the same arguments and return the same values. Prepare-update method $t(a)$ applies $u(a)$ on a copy of the current replica state s resulting temporary state $s' = s \bullet u(a)$ and transmits it further to the client. Here, this state is used as an argument for the effect-update method $u'(s')$ which delivers it to all replicas by the underlying protocol of CmRDT and merges it using the original CvRDT merge method: $s \bullet u'(s') = s \bullet m(s')$. Note that the new update does a merge and, since merge operations commute for CvRDT, then the update also commutes. This fact combined with the reliable communication channel assumption gives us a CmRDT.

2.7.2 State-based Emulation of an Op-based Object

In order to prove that a SEC op-based object can be emulated as a SEC state-based object, we consider a CmRDT represented by the tuple (S, s^0, f). This can be emulated by a CvRDT object $((S \times U \times U), \sqsubseteq, (s^0, \varnothing, \varnothing), f')$ described next. CvRDT's new state is now defined as (s_m, M, D), where s_m is the state of the corresponding original CmRDT. Again, queries can be ignored because they pose no problems, $(s_m, M, D) \bullet q'(a) \overset{def}{=} (s_m \bullet q(a), M, D)$. For updates, whenever an op-based one is called, it is added to a set M of received messages to be delivered. In order to avoid duplicate deliveries, messages which are delivered are stored in a set D. Thus, M and D are grow-only sets. The $\sqsubseteq$ relation is defined by $(s_m, M, D) \sqsubseteq (s'_m, M', D') \overset{def}{=} M \subseteq M' \wedge D \subseteq D'$. An update $u'(a)$ first calls the original prepare-update method $t(a)$ of the CmRDT and then the effect-update $u(a)$ locally. It then uses the recursive function d to process updates:

$$d(s_m, M, D) \overset{def}{=} \begin{cases} d(s_m \bullet u(a), M, D \cup \{u(a)\}), & \text{if } \exists u(a) \in M \setminus D \\ (s_m, M, D), & \text{otherwise} \end{cases}$$

Intuitively, d processes any pending updates, i.e. in M but not in D. Therefore, $u'(a)$ will be defined as $(s_m, M, D) \bullet u'(a) \overset{def}{=} d(s_m \bullet t(a), M \cup \{u(a)\}, D)$, provided that the update's downstream precondition is true. Merge operations take the union of received messages and process available updates: $(s_m, M, D) \bullet m(s'_m, M', D') \overset{def}{=} d(s_m, M \cup M', D)$.

It should be noted that the states of the emulating CvRDT object form a monotonic semilattice in the domain $S \times U \times U$. Calling or delivering an operation adds it to the corresponding set and thus advances the state in the partial order. The *merge* method is defined as the union of sets M and possibly D, and is therefore a LUB operation. After a mutual merge, the D sets in both replicas are identical and their states are equivalent. Also, M contains non-concurrent updates in causal order, while concurrent updates may appear in any order.

2.8 Examples of CRDTs

This section gives some concrete examples of CRDTs which are known in the literature [7]. Any of them can be specified using either the state-based or the op-based approach.

2.8.1 Counters

The implementation for distributed replicated counters is inspired by vector clocks. The idea is to consider the counter as a state-based object whose payload is a vector of integers with the same number of elements as replicas of that counter, n. Thus, the domain where the payload take its values from is $S = \mathbb{N}^n$ and the initial state for all replicas is $s^0 = [0, \ldots, 0]$. The update operation $inc(i)$ increments the i^{th} component of the payload, where each process p_i is allowed to increment only its own index i: if $s = [s[0], \ldots, s[i], \ldots, s[n-1]]$, then $s \bullet inc(i) = [s[0], \ldots, s[i]+1, \ldots, s[n-1]]$. The query $value()$ returns $|s| \overset{def}{=} \sum_i s[i]$, while the *merge* method computes the per-index maximum of the two vectors: $s \bullet m(s') = [max(s[0], s'[0]), \ldots, max(s[n-1], s'[n-1])]$. If we consider the order operation between these vectors as $s \sqsubseteq s' \iff s[i] \leq s'[i], \forall i \in \{0, \ldots, n-1\}$, then $(S, \sqsubseteq)$ forms a monotonic semilattice and merge produces the LUB. Thus this **increment-only** counter is a CvRDT.

In order to support decrement operations, two increment-only counters, I and D, can be grouped. Incrementing does an increment on I, while decrementing does an increment in D. Now, *value* will return $|I|-|D|$ and m will merge the corresponding counters: I with I' and D with D'. The new partial order is given by $\sqsubseteq$ defined as $(I, D) \sqsubseteq (I', D') \overset{def}{=} I \sqsubseteq I' \wedge D \sqsubseteq D'$.

2.8.2 Registers

A register is a memory cell storing a value of any data type and supports update operation *assign* to change this value and query operation *value* to retrieve it. Non-concurrent *assign*s are consistent with sequential semantics: one overwrites the previous one. For concurrent updates, there are two approaches: either to keep just one (LWW-Register), or to keep both (MV-Register).

A **Last-Writer-Wins Register** (**LWW-Register**) associates a unique timestamp to each update to create a total order of assignments consistent with causal order. This means that for any two assignments, if one happens before the other, then the first one's timestamp is less than the second one's. Timestamps can be implemented by concatenating a replica unique ID (such as MAC address) with a local counter. The usual supported state-based operations are defined as: the *value* query returns the current value, the *assign* update assigns a new value to the payload and generates a new associated timestamp, the *merge* procedure selects the value with the maximum timestamp. The equivalent op-based *assign* generates the timestamp at the source in the prepare-update phase and changes the value in the effect-update phase only if the new timestamp is greater than the old one. This type of register is found in many distributed systems. One example is NFS [12], where the LWW-Register represents a file (or a file block).

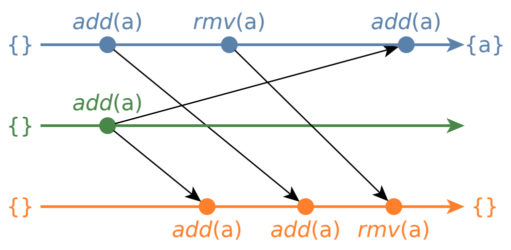


Figure 2.3: Counter-example for a CRDT set

The **Multi-Value Register** (**MV-Register**) uses a version vector instead of a timestamp to identify each assignment. The state-based *assign* method generates a new version vector that is greater than all existing ones and adds the pair consisting of the new value and the new version vector to the payload set. The *merge* operation computes the union of every element in the first set that is not dominated by an element in the second set. Examples of this type can be found in the Coda file system [13] and Amazon's Dynamo [5].

2.8.3 Sets

Sets constitute the building blocks for almost all complex data structures: containers, maps, and 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. Consider the example in Figure 2.3 of a 3-times replicated op-based set. Initially the set is empty. Replica 1 adds element a and then removes it. Replica 2 adds the same element a and sends the update to Replica 1, thus making $\{a\}$ the final state at Replica 1. Replica 3 receives both *add* updates from Replica 1 and Replica 2, but only the first one has effect (adding the same element twice in a set does not change the state). In the end, Replica 3 receives also the *remove* update from Replica 1, making $\varnothing$ the final state here. In this example we can see that although Replica 1 and Replica 3 both applied the operations in causal order, their states diverge.

Hence any CRDT implementation of a set can only approximate the sequential set. The difference is given by having to choose which operation to take precedence in a concurrent $add(e) \parallel remove(e)$ situation: either *remove* wins (the 2P-Set), or *add* wins (the OR-Set).

The first CRDT implementation of a set, **Grow-Only Set** (**G-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 $merge(S, T) =$

Specification 1 G-Set (state-based)

```
1: payload $A = \varnothing$
2: update $add(e)$
3:     $A \coloneqq A \cup \{e\}$
4: query $lookup(e)$ : boolean
5:     return $e \in A$
6: compare $(S, T)$ : boolean
7:     return $S.A \subseteq T.A$
8: merge $(S, T)$ : payload
9:     return $S.A \cup T.A$
```

Specification 2 2P-Set (state-based)

```
1: payload $A = \varnothing, R = \varnothing$
2: update $add(e)$
3:     $A \coloneqq A \cup \{e\}$
4: update $remove(e)$
5:     pre $lookup(e)$
6:     $R \coloneqq R \cup \{e\}$
7: query $lookup(e)$ : boolean
8:     return $e \in A \wedge e \notin R$
9: compare $(S, T)$ : boolean
10:     return $S.A \subseteq T.A \vee S.R \subseteq T.R$
11: merge $(S, T)$ : payload
12:     let $U.A = S.A \cup T.A$
13:     let $U.R = S.R \cup T.R$
14:     return $U$
```

$S \cup T$ computes the LUB in the monotonic semilattice $(S, \sqsubseteq)$. And so, G-Set is also a CvRDT.

A **Two-Phase Set** (**2P-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 adding and set R for removing. Adding or removing the same element twice or adding an already removed element has no effect. The *merge* operation computes the LUB of the two sets by applying set-union on each one individually. Thus, the state-based 2P-Set is a CvRDT. Considering the op-based approach, concurrent *add* and *remove* operations on same element and concurrent operations on different elements commute. Also $add(e) \parallel add(e)$ and $remove(e) \parallel remove(e)$ commute by definition. It follows that 2P-Set is also a CmRDT.

If we guarantee that each element in the set is unique[1] and that an $add(e)$ is deliv-

[1] To achieve global unique elements across all replicated sets, Lamport clocks can be used or a

Specification 3 U-Set (op-based)

```
1:  payload $S = \varnothing$
2:  query $lookup(e)$ : boolean
3:      return $e \in S$
4:  update $add(e)$
5:      prepare($e$)
6:          pre $e$ is unique
7:      effect($e$)
8:          $S := S \cup \{e\}$
9:  update $remove(e)$
10:     prepare($e$)
11:         pre $lookup(e))$
12:     effect($e$)
13:         pre $add(e)$ has been delivered
14:         $S := S \setminus \{e\}$
```

ered 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** and it is presented in Specification 3. It is easy to show that the U-Set is a CmRDT: since each element is unique, *add*s are independent; also there cannot be concurrent *add* and *remove* on the same element because each *remove* must casually follow the corresponding *add*.

An alternative solution, called a **PN-Set**, 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).

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** (**OR-Set**) introduced in [14] is closer to the usual set semantics. The idea is to uniquely tag each added element. When removing an element, only associated tags observed at the source are removed.

Specification 4 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

function which generates random numbers from a large space.

Specification 4 OR-Set (op-based)

```
 1: payload $S = \varnothing$
 2: query $lookup(e)$ : boolean
 3:     return $\exists u : (e, u) \in S$
 4: update $add(e)$
 5:     prepare($e$) : tag
 6:         let $\alpha = unique()$
 7:         return $\alpha$
 8:     effect($e, \alpha$)
 9:         $S := S \cup \{(e, \alpha)\}$
10: update $remove(e)$
11:     prepare($e$) : set
12:         pre $lookup(e)$
13:         let $R = \{(e, u) \mid \exists u : (e, u) \in S\}$
14:         return $R$
15:     effect($R$)
16:         pre $\forall (e, u) \in R : add(e, u)$ has been delivered
17:         $S := S \setminus R$
```

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 $add(e) \parallel remove(e)$ will give precedence to $add(e)$, in contrast to the 2P-Set.

This behavior is illustrated in Figure 2.4. Here the two $add(a)$ operations generate unique tags α and β, respectively. When Replica 1 applies $rmv(a)$, it translates to removing (a, α) downstream. The $add(a)$ at Replica 2 is concurrent to the $rmv(a)$ at Replica 1 and, therefore, (a, β) remains in the final state.

OR-Set is a CmCRDT because: i) concurrent *add*s commute since each one is unique; ii) concurrent *remove*s commute since any common pairs have the same effect, and any disjoint pairs have independent effects; iii) concurrent $add(e_1)$ and $remove(e_2)$ commute: if $e_1 \neq e_2$ they are independent, else *remove* has no effect.

The state-based approach is presented in Specification 5. 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

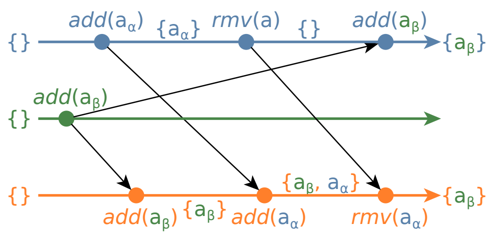


Figure 2.4: OR-Set (op-based)

Specification 5 OR-Set (state-based)

```
1: **payload** $A = \varnothing, R = \varnothing$
2: **query** $lookup(e)$ : boolean
3:     **return** $\exists\alpha : (e,\alpha) \in A \land \nexists\beta : (a,\alpha,\beta) \in R$
4: **update** $add(e)$
5:     **let** $\alpha = unique()$
6:     $A \coloneqq A \cup \{(e,\alpha)\}$
7: **update** $remove(e)$
8:     **pre** $lookup(e)$
9:     **let** $\beta = unique()$
10:    **let** $R' = \{(e,\alpha,\beta) | \exists(e,\alpha) \in A\}$
11:    $R \coloneqq R \cup R'$
12: **compare** $(S,T)$ : boolean
13:    **return** $S.A \subseteq T.A \land S.R \subseteq T.R$
14: **merge** $(S,T)$ : payload
15:    **let** $U.A = S.A \cup T.A$
16:    **let** $U.R = S.R \cup T.R$
17:    **return** $U$
```

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. The partial order on the payload states is given by the relation $S \sqsubseteq T \iff S.A \subseteq T.A \land S.R \subseteq T.R$. Because update operations always compute larger states relative to $\sqsubseteq$ and because *merge* computes the LUB, state-based OR-Set is also a CRDT.

2.8.4 Graphs

A graph is defined as a pair of sets (V,E), where V is the set of vertices and $E \subseteq V \times V$ is the set of edges between vertices. For V and E any of the set structures previously described can be used. Since we cannot add an edge if any of the two connecting vertices is missing and we cannot remove a vertex if it supports an edge, update operations on vertices and edges are not independent. Therefore an analysis is needed when $addEdge(u,v) \parallel removeVertex(u)$. There are three possibilities: i) give precedence to $removeVertex(u)$: all edges to or from u are also removed, ii) give precedence to $addEdge(u,v)$: if either u or v has been removed, it is restored, and iii) $removeVertex(u)$ is delayed until all concurrent $addEdge(u,v)$ operations have been executed.

Since option ii) has complex semantics and option iii) requires synchronization, option i) is presented in Specification 6. The idea is to use the same approach as for the OR-Set. The payload contains two sets, one for vertices and one for edges. Adding a vertex v first generates a unique identifier α at the source and then adds

Specification 6 Directed graph (op-based)

```
**payload** $V = \varnothing, E = \varnothing$
**query** $lookupVertex(v)$ : boolean
    **return** $\exists \alpha : (v, \alpha) \in V$
**query** $lookupEdge((v', v''))$ : boolean
    **return** $lookupVertex(v') \wedge lookupVertex(v'') \wedge \exists \alpha : ((v', v''), \alpha) \in E$
**update** $addVertex(v)$
    **prepare**$(v)$ : tag
        **let** $\alpha = unique()$
        **return** $\alpha$
    **effect**$(v, \alpha)$
        $V := V \cup \{(v, \alpha)\}$
**update** $removeVertex(v)$
    **prepare**$(v)$ : set
        **pre** $lookupVertex(v)$
        **pre** $\nexists v' : lookupEdge((v, v'))$
        **let** $R = \{(v, \alpha) | \exists \alpha : (v, \alpha) \in V\}$
        **return** $R$
    **effect**$(R)$
        $V := V \setminus R$
**update** $addEdge(v', v'')$
    **prepare**$(v', v'')$ : tag
        **pre** $lookupVertex(v')$
        **let** $\alpha = unique()$
        **return** $\alpha$
    **effect**$(v', v'', \alpha)$
        $E := E \cup \{((v', v''), \alpha)\}$
**update** $removeEdge(v', v'')$
    **prepare**$(v', v'')$ : set
        **pre** $lookupEdge((v', v''))$
        **let** $R = \{((v', v''), \alpha) | \exists \alpha : ((v', v''), \alpha) \in E\}$
        **return** $R$
    **effect**$(R)$
        $E := E \setminus R$
```

the pair (v, α) to the set of vertices in the downstream phase. To remove a vertex v, the prepare-update method computes the set of pairs that contain v at source and the effect-update method removes this set from the set of vertices in all replicas. Since we have the guarantee that operations are delivered in the causal order, we know that for each element to be removed the corresponding *addVertex* has been called. As with OR-Sets, *addVertex*(v) wins when *addVertex*$(v) \parallel$ *removeVertex*(v) because *removeVertex*(v) operates only on the locally observed vertices at source. The same approach is used for adding and removing edges. In order to avoid the anomalies mentioned in the beginning of the section, preconditions are used in the prepare phases.

To prove that this op-based specification is a CRDT, it should be shown as usually that concurrent updates commute. $addVertex(v') \parallel addVertex(v'')$ commute as both generate unique tags and perform a set union, which is a commutative operation. $removeVertex(v') \parallel removeVertex(v'')$ commute because they perform $V \setminus R'$ and $V \setminus R''$, respectively, and $V \setminus R' \setminus R'' = V \setminus R'' \setminus R'$, where R', R'' are the sets of observed vertices to be removed. $addVertex(v') \parallel removeVertex(v'')$ also commute because $addVertex(v')$ generates a fresh unique α, and therefore $(v', \alpha) \notin R''$ and $V \cup \{(v', \alpha)\} \setminus R'' = V \setminus R'' \cup \{(v', \alpha)\}$. For edges, the proof follows the same steps. Finally, since any concurrent updates on vertices and edges modify disjoint internal sets, from this it results that they commute too.

As seen, graphs are more complex data structures, but they can be easily implemented by composing simpler CRDTs, like OR-Sets or 2P-Sets. However, maintaining a particular shape, such as a tree or a directed acyclic graph (DAG), cannot be done by a CRDT since this requires synchronization to ensure a global acyclicity invariant. But types which employ some stronger forms of acyclicity are viable. In a **monotonic DAG** an edge may be added only if it is oriented in the same direction as an existing path, thus strengthening the partial order defined by the DAG. The specification for this type can be found in [7].

2.9 Other Approaches to Consistency

The fundamental principles on database replication are laid out in [15] 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 [16]. 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 [17]. A survey on other models is presented in [18].

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 [12], AFS [13] and Coda [19] file systems and in key-value stores such as Amazon's Dynamo [5] or Riak [20]. The mathematical foundations were laid by Baquero and Moura [21] and later extended by Shapiro and Preguiça on their work on Treedoc [22] 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 [10] and the IceCube cooperative system [14].

Later, a formal definition and rigorous system model for CRDT were published in [7] and [8]. These are the first works to engage a comprehensive and systematic study on CRDTs. The thesis extends these definitions to support an efficient synchronization algorithm which transmits only deltas of updates and introduces a specification for partitioning a CRDT OR-Set replica into disjunctive subsets. A garbage collection mechanism to reclaim obsolete elements from the set is also discussed. On the practical side, to the author's 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 [23, 24], but they focus only on testing the specifications for CRDTs locally and in-memory, without a real database store support. ConcoRDanT [25] is an interesting project aimed to investigate the principles of these types.

2.10 Short Introduction on Redis

NoSQL is a class of database management systems introduced as an alternative to traditional RDMBSs in order to cope with problems of certain data models, such as low throughput, unneeded complexity, and horizontal scalability [26]. Representatives of this class of applications compensate the lack of advanced features with gains in scalability and performance.

In this sense, Redis [6] is a widely used, open-source, in-memory, key-value store with many features which make it an ideal database system for evaluating the contributions of this thesis as shown in Chapter 4. Its data model is a dictionary that maps keys to values. All the following data types are supported:

- *Strings*: the most basic data type, binary safe and can contain any kind of data. Redis also allows a number of interesting operations to be applied to strings: consider them as integers and increment or decrement them atomically, append to them, use them as random access vectors to get a substring or encode a lot of binary data and accessing bits of it at given offsets.
- *Lists*: simply lists of strings, sorted by insertion order. Accessing elements is very fast near the extremes of the list, $\mathcal{O}(1)$ at the head (on the left) or at the tail (on the right), but is slow in the middle of a very big list, $\mathcal{O}(N)$.
- *Sets*: unordered non repeating collection of strings. They support operations to add, remove, and test for existence of members in $\mathcal{O}(1)$ time.
- *Sorted sets*: similar to sets but where every element is associated to a floating number score and the elements are sorted by this score. Usual operations take $\mathcal{O}(log(N))$ time.
- *Hashes*: maps between string fields and string values, can also be seen as collections of (field, value) pairs. They are suitable for storing objects: the key can represent the object id, while the fields can represent the attributes.

The keys in Redis can be only strings, while as values all the above data types are supported. Other Redis features include persistence, replication, transactions, pipelining, or Lua scripts. In addition, Redis has support for associating timeouts with keys. After the timeout has expired, the key is automatically deleted. Some of these features match perfectly with the design from Chapter 3. For a complete documentation, the reader is referred to the Redis website [6].

Chapter 3

Design of Sharded OR-Set

The section presents the main contributions of this work: improvement to the state-based OR-Sets specification to transfer only deltas between replicas instead of full states, called *delta-based synchronization (merging)* algorithm[1], and support for *sharding* in the sense that each replica can be split into disjunctive subsets and stored in a cluster. This data type with sharding and delta-based merging is called **Sharded OR-Set** (**SOR-Set**). Lastly, a garbage collection mechanism which maintains the CRDT properties is given.

3.1 The Case for a SOR-Set

In order to protect resources against various external attacks, such as denial-of-service, companies usually employ filters on different levels of their infrastructure [27]. One such type of filters keeps track and limits the number of events allowed for a given IP address or account, such as login attempts, password changes, emails sent, and so on. To this purpose, there is a need for data structures to provide low-latency, high-throughput for read and write operations. Furthermore, facilities are required to accumulate statistics on these events in local data centers and later to synchronize the geo-replicated databases. CRDTs fit very well in this context by providing means to achieve the aforementioned demands. One usage example could be to keep in a CRDT counter the number of login attempts from one IP address and in a CRDT set the associated unique passwords. Given the modular nature of CRDTs, this use case can be also extended to a graph-like structure in order to store relations among various events and entities, such as account, IP addresses, aliases, and login attempts. The focus of this thesis is the CRDT set type.

As seen in Section 2.8, there are different ways of constructing a CRDT set data

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

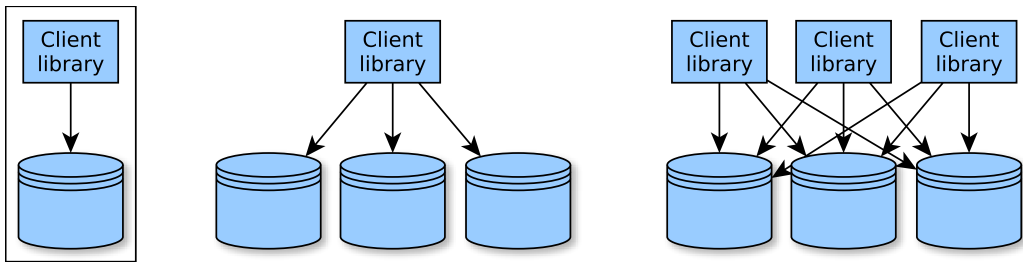


Figure 3.1: Use-case scenarios for a sharded CRDT set

structure. They mainly differ in which operation has priority in an *add*(e) $\parallel$ *remove*(e) situation. OR-Sets give precedence to *add* and conform to the sequential specification of a set. Moreover, OR-Sets are very intuitive and do not suffer from the semantics anomalies encountered in the other set specifications: a removed element can never be added again (2P-Set) or adding an element to an empty set keeps it empty (PN-Set).

Section 2.7 showed that if a CRDT can be implemented using any of the two approaches, it can also be implemented using the other. However, there are advantages and disadvantages to each one. The goal is to have the simplicity of state-based constructs corroborated with the transfer efficiency of the op-based ones. We know that for state-based CRDTs, in order to merge the states of two replicas, we have to transfer the state from one location to the other and then apply the *merge* method. Both these operations are costly in terms of computation time and network traffic. If this behavior can be improved while ensuring that updates still advance the states in the monotonic semilattice and that merging still gives the LUB (and thus CRDT properties are maintained), then this approach becomes very attractive. Also, by choosing the state-based variant, there are no more restrictions on the communication channel specific to the op-based approach: a reliable broadcast is not needed and updates can be lost along the way or applied multiple times.

Access to the replicated set type is envisioned in this thesis through a client library which can be deployed in any of the three scenarios from Figure 3.1. The first one corresponds to an asynchronous cache where both library and database are stored on the same machine. Caches from different machines are made coherent through CRDT-specific synchronization mechanisms. Next two cases scale in size and performance and by sharding the database across different machines and by deploying many clients to load balance incoming requests. Implementation details and evaluation results of this library are presented in Chapter 4.

Finally, the flexibility of CRDTs allows to select different synchronization topologies. One example would be a hierarchical representation where one could aggregate data in the replica leaves and then propagate updates towards the root replica, as needed. Another example would be to eventually distribute all updates in the system according to any gossip or anti-entropy protocols [9, 10]. A combination of these two examples is given in Figure 3.2. Here each data center aggregates and

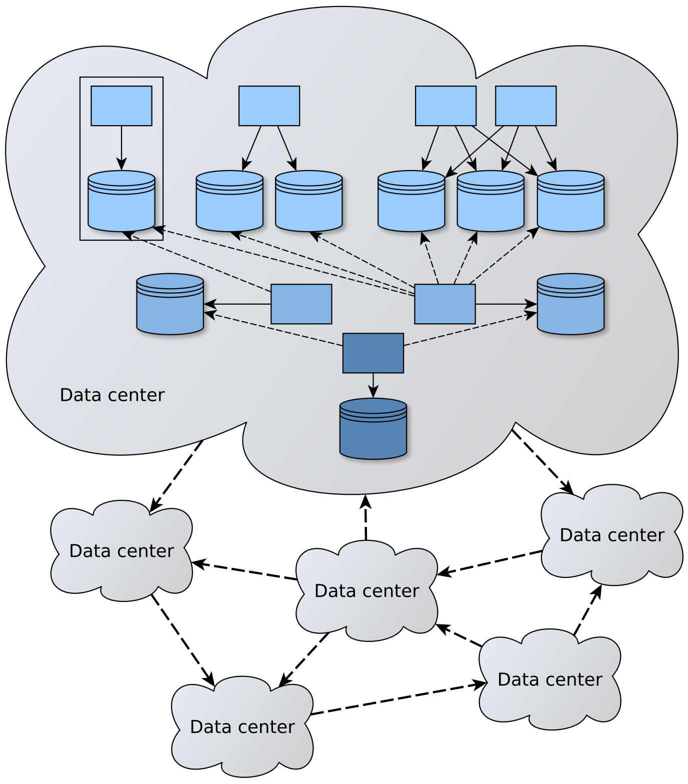


Figure 3.2: Example of synchronization topology

internally merges event statistics in a tree-like manner, while the distribution of updates across all data centers is done over a mesh topology using an anti-entropy protocol. As shown next in this chapter, stores and data centers can operate during network partitioning. Many other use cases can be envisioned since synchronization of CRDTs is highly customizable: one can decide for each replica in part when and where to pull updates from.

3.2 A Delta-based Synchronization Algorithm

A design for a state-based OR-Set which employs a delta-based algorithm for merging replica states is given in Specification 7. In addition to the original state-based OR-Set which had sets A and R for added and, respectively, removed elements, 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. For example, if there are N replicas, the *replica* function can simply generate a number in the range $[0, \ldots, N-1]$ and T can be an integer array of size N. Alternatively, *replica* can generate a string identifying the replica and T in this case will be a hash

Specification 7 OR-Set with delta-based synchronization (state-based)

```
payload $A = \varnothing, R = \varnothing, T = [\,]$
query $lookup(e)$ : boolean
    return $\exists(e,t,r) \in A \land \nexists(e,t,r,t',r') \in R$
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 $S_1.A \subseteq S_2.A \land S_1.R \subseteq S_2.R \land S_1.T[i] \leq S_2.T[i], \forall i$
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 $P$
```

table with string keys and integer values.

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.

Remember that the *merge* method in the traditional OR-Set computed the union between the As and between the Rs. For any two sets X and Y, it is known that the following holds: $X \cup Y = X \cup (Y \setminus X)$. Since the goal is to transfer only the difference between sets from one replica to the other and not the whole set, the right-hand side formula is used. This is done with the help of timestamp vector T. To exemplify, let us consider that we want to merge the state of Replica 2 into Replica 1. In the first step, Replica 1 sends its T to Replica 2. Replica 2

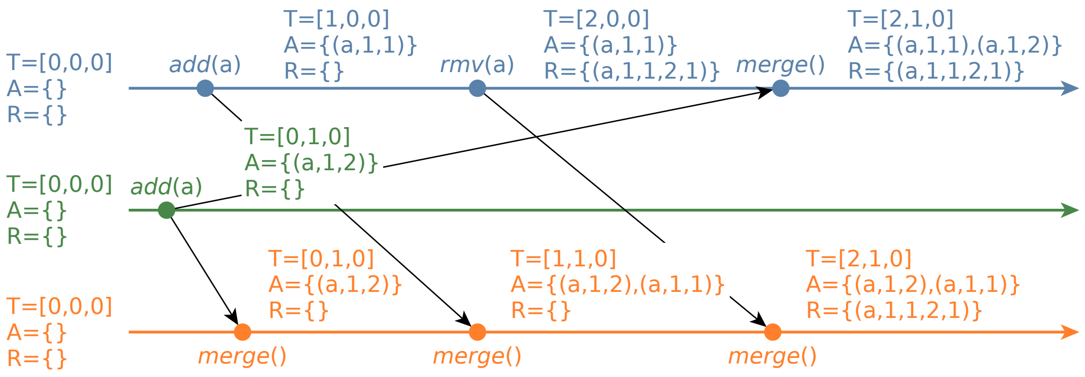


Figure 3.3: OR-Set with delta-based synchronization (state-based)

then computes the updates based on this vector: missing tuples are those added or removed at Replica 2 which are not present in Replica 1, i.e. the logical clock is less than the tuple's timestamp: $A' = \{(e, t, r) \in S_2.A \mid S_1.T[r] < t\}$. This set of updates together with Replica 2's timestamp vector are sent back to Replica 1. Finally, Replica 1 inserts the updates in the corresponding sets and then updates its timestamp vector with the maximum between the current one and the received one component-wise: $P.T = max(S_1.T, S_2.T)$. This last operation is required so that the next *merge* will only ask for the updates after this time.

Thus, the timestamp vector helps to compute the difference between two sets by filtering based on the tuples timestamp. In this sense, it acts as a vector clock [28] which guarantees the partial order between updates. Also, because of the transitivity property of the vector clock, each $merge(S_1, S_2)$ includes 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.

Figure 3.3 shows how the states evolve in this new OR-Set when applying the same operations as in the previous example from Figure 2.4.

Proof that delta-based synchronization maintains the CRDT properties. In order to prove that this construct is a CRDT, 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, it was shown above that *merge* basically computes the union of the added and, respectively, removed sets and the maximum of the two timestamp vectors. Hence we have $merge(S_1, S_2) = S_1 \sqcup S_2$ (LUB) which concludes the proof. □

3.3 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. Therefore, each replicated set can reside in a cluster, as illustrated in Figure 3.4. 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 is used which forwards the usual *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. It does not matter where the client resides as long as it knows how to communicate with both clusters. For example, one way is for the client to know the topology and to talk to each shard. Another way would be to use a DHT overlay for each cluster which the client can bootstrap onto. Then the client request would bounce in the overlay network until it reaches the shard which is responsible for that element. For further details about the architecture, the reader is referred to Section 4.1.

Specification 8 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 *timestamp "matrix"* T is used. This is actually a degenerated matrix which has as many rows as there are clusters and which on row rc_i has $|rc_i|$ columns. Similar to the OR-Set with delta-based synchronization, each cell $T[rc][rs]$ stores the latest version of the logical clock of the shard (rc, rs). Each shard has its own T matrix. Since when adding or removing an element from the source replica, it will be added or, respectively, removed from only one shard (rc, rs) (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).

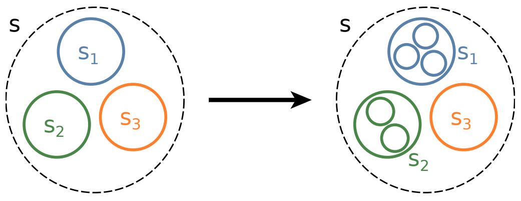


Figure 3.4: Sharding of OR-Sets

Specification 8 SOR-Set with delta-based synchronization (state-based)

```
1:  payload $A_i^j = \varnothing, R_i^j = \varnothing, T_i^j = [\,][\,], \forall j \in \{1, \ldots, |rc_i|\}$
2:                                      $\triangleright$ $rc_i$ - Replica cluster $i$; $rs_i^j$ - Replica shard $j$ of $rc_i$
3:  query $lookup_i(e)$ : boolean
4:      let $j = hash_i(e)$
5:      return $\exists(e, t, rc, rs) \in A_i^j \wedge \nexists(e, t, rc, rs, t', rc', rs') \in R_i^j$
6:  update $add_i(e)$
7:      let $j = hash_i(e)$
8:      let $rc = rc_i$
9:      let $rs = rs_i^j$
10:     let $t = T_i^j[rc][rs] + 1$
11:     $A_i^j := A_i^j \cup \{(e, t, rc, rs)\}$
12:     $T_i^j[rc][rs] := t$
13: update $remove_i(e)$
14:     pre $lookup_i(e)$
15:     let $j = hash_i(e)$
16:     let $rc' = rc_i$
17:     let $rs' = rs_i^j$
18:     let $t' = T_i^j[rc'][rs'] + 1$
19:     $R_i^j := R_i^j \cup \{(e, t, rc, rs, t', rc', rs') \mid \exists(e, t, rc, rs) \in A_i^j\}$
20:     $T_i^j[rc'][rs'] := t'$
21: compare $(rc_x, rc_y)$ : boolean
22:     let $\tilde{T}_x = version(rc_x)$
23:     let $\tilde{T}_y = version(rc_y)$
24:     return $(\bigcup_j A_x^j \subseteq \bigcup_k A_y^k) \wedge (\bigcup_j R_x^j \subseteq \bigcup_k R_y^k) \wedge (\tilde{T}_x \leq \tilde{T}_y)$
25:                                      $\forall j \in \{1, \ldots, |rc_x|\}; \forall k \in \{1, \ldots, |rc_y|\}$
26: merge $(rc_x, rc_y)$ : payload
27:     let $\tilde{T}_x = version(rc_x)$
28:     let $\tilde{T}_y = version(rc_y)$
29:     $\forall j \in \{1, \ldots, |rc_y|\}$
30:         let $A' = \{(e, t, rc, rs) \in A_y^j \mid \tilde{T}_x[rc][rs] < t\}$
31:         let $R' = \{(e, t, rc, rs, t', rc', rs') \in R_y^j \mid \tilde{T}_x[rc'][rs'] < t'\}$
32:     $\forall j \in \{1, \ldots, |rc_x|\}$
33:         let $Z.A_x^j = A_x^j \cup \{(e, t, rc, rs) \in A' \mid j = hash_x(e)\}$
34:         let $Z.R_x^j = R_x^j \cup \{(e, t, rc, rs, t', rc', rs') \in R' \mid j = hash_x(e)\}$
35:         let $Z.T_x^j = max(T_x^j, \tilde{T}_y)$
36:     return $Z$
```

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 timestamp matrix for shard rs_i^j. Adding an element e at replica source i has to first determine in which shard it should be stored by calling $hash_i(e)$. Each cluster may have its own distinct partitioning function. The procedure then increments the logical clock of that shard and inserts the tuple (e, t, rc_i, rs_i^j) into the set of added elements A_i^j. Removing an element e follows the same principle as before: locally observed elements e are tagged and added to the remove set R_i^j. An element will be in the set if it is in A_i^j and not in R_i^j.

Merging the state of a remote replica from cluster rc_y into the local state in cluster rc_x proceeds as follows. First, it computes $\tilde{T}_x = version(rc_x)$ using the formula:

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

What the *version* function does is to determine a minimum version for the whole cluster by combining the information from all timestamp matrices T. This is needed because clusters may have different sizes and thus T_x^j does not necessarily have a correspondent T_y^j. The version is basically computed by choosing the minimum from all T_x^j component-wise, except for the row rc_x, where the maximum is chosen instead. Updating the set i changes only row rc_i in all T_i^j since each shard increments its own counter only. Therefore, after m updates to shard rs_i^j, assuming no synchronization takes place, $T_i^j[rc_i][rs_i^j] = m$ and $T_i^j[rc_i][rs_i^k] = 0, \forall k \neq j$, while the rest of the cells remain unaffected. For this reason, on row rc_x of $\tilde{T}_x$, the maximum value has to be chosen. The procedure is identical for remote cluster rc_y.

The next step in the delta-based synchronization algorithm is to compute the updates in rc_y which are missing in rc_x. For this, $\tilde{T}_x$ is used to apply the same principle as before: missing updates are those whose timestamp is greater than the corresponding logical clock of the local cluster. Sets A' and R' gather all these updates from all shards in the remote cluster rc_y. What remains is to distribute them in rc_x according to the $hash_x$ function and to update the timestamp matrix of all shards rs_x^j.

Lastly, some important observations should be made regarding the SOR-Set. First, it is still an OR-Set at the core, which means it has the same semantics: *add*(e) operation has precedence when *add*(e) $\parallel$ *remove*(e) and *remove*(e) removes locally observed elements e. Second, it employs the delta-based synchronization algorithm introduced in Section 3.2. Third, 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 matrix $\tilde{T}_y$ 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. The last observation to make is that 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 row 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 row rc_y and missed updates will be fetched with the next merge after the shard restores. These failure scenarios are tested and discussed in Section 4.4.

Proof that sharding maintains the CRDT properties. Consider a replica state as $s_i = (A_i, R_i, \tilde{T}_i)$, where $A_i = \bigcup_{j\in\{1,\ldots,|rc_i|\}} A_i^j$, $R_i = \bigcup_{j\in\{1,\ldots,|rc_i|\}} R_i^j$, and $\tilde{T}_i$ was defined above. 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, which concludes the proof. □

3.4 Garbage Collection

As seen in Specification 8, 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 configured based on application semantics. For example, we may not be interested in IP addresses for logging into a given user account which are older than one month.

Any $(e, t, rc, rs) \in A$ from the SOR-Set specification 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 tuple expiration can now be formulated.

Theorem 3 (Tuple 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 garbage collection while maintaining the lookup semantics. *Garbage* which can be reclaimed refers to all the expired tuples in the store. The idea is to 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 and can be expressed in any time unit, e.g. minutes, hours, days, etc. 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 [6] or Cassandra [29] 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.

Thus, set A will contain pairs $\langle(e, t, rc, rs), TTL(e)\rangle$ and set R will contain pairs $\langle(e, t, rc, rs, t', rc', rs'), TTL(e)\rangle$. To ensure that tuples corresponding to e expire in the order in which they were added, it is sufficient to stamp them with the same value $TTL(e)$. This initial value will gradually decrease until it reaches 0 and then the tuple can be removed. When copying the tuples to other shards, their remaining TTL is 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.

Because it is possible for $ADD(e)$ to be added on a shard, copied to another shard, and then here a $RMV(e)$ inserted, all replicas are required to use the same $TTL(e)$ function for stamping tuples of a particular element e. However, 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. This happens because time freezes for copies while they are in transit and their TTLs remain constant. However, this does not invalidate the lookup semantics according to the tuple expiration theorem.

Proof that garbage collection maintains the CRDT properties. In order to prove that this SOR-Set construct together with the garbage collection mechanism described above retains 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 8: $S_i = \bigcup_{\forall j \in \{1,\ldots,|rc_i|\}} S_i^j$, where S is A or R. From this point, the proof follows the same rationale as for the SOR-Set. □

Chapter 4

Implementation and Evaluation

This chapter gives the implementation details of a Redis client library for the SOR-Set. The purpose is to give an interface to the SOR-Set by connecting to a replica cluster and providing access to the usual set operations: adding, removing, looking up an element, or asking to synchronize with other replica clusters. The next sections include the architectural design, a Redis database schema for storing the tuples sets and timestamp matrix, the pseudocode for all set operations, and finally the results obtained on evaluating the library.

4.1 Architecture

The payload for each shard (sets of added and removed tuples and the timestamp matrix) is stored in a separate Redis database, or *store*. The terms shard and store will be used interchangeably from now on. Figure 4.1 illustrates an example of two replicas of a set, each one stored in a different cluster: first one in cluster with id $rc = A$ which is sharded on 3 stores and second one in cluster with id $rc = B$ sharded on 2 stores. A shard is identified by an IP address and a port number the Redis server is listening on. The collection of all replica clusters together with their corresponding shards will be referred to as the *topology*. An example of topology in XML format for the deployment in Figure 4.1 is given in Figure 4.2. Because a stateless client library is desired, everything is stored in the Redis database, including the topology.

The client's interface is given by the the following methods: $\mathit{boot}(\mathit{ip}, \mathit{port})$, $\mathit{add}(e)$, $\mathit{remove}(e)$, $\mathit{lookup}(e)$, and $\mathit{merge}(rc)$. First method is used for bootstrapping to a cluster representing the replica of the set we want to operate on. This is done by connecting to any store/shard in the cluster and fetching the topology. Thus, when connected, the client has the addresses of all the shards in the network. However, for adding, removing and looking up elements, it talks only to the shards in the cluster

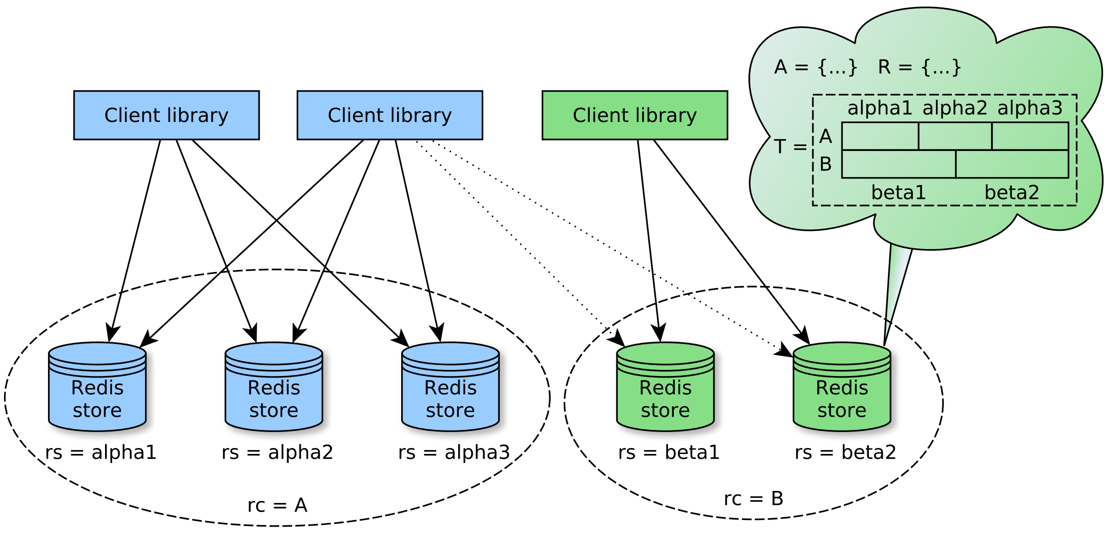


Figure 4.1: Client library architecture

```
<topology>
  <cluster id=”A”>
    <store id=”alpha1” ip=”127.0.0.1” port=”6379”/>
    <store id=”alpha2” ip=”127.0.0.1” port=”6382”/>
    <store id=”alpha3” ip=”127.0.0.1” port=”6383”/>
  </cluster>

  <cluster id=”B”>
    <store id=”beta1” ip=”127.0.0.1” port=”6380”/>
    <store id=”beta2” ip=”127.0.0.1” port=”6384”/>
  </cluster>
</topology>
```

Figure 4.2: Topology example in XML

it booted onto, or more specific to the shard given by the hash function $hash_i(e)$. The addresses of the stores in the other clusters are needed for the *merge* method (depicted with dotted lines in Figure 4.1). To any cluster we can connect as many clients as we wish. Here there are two clients working with Replica A and one client working with Replica B.

4.2 Database Schema

Listing 1 contains the Redis schema for storing the data on each shard. Underlined words represent hard-coded strings, whereas non-underlined ones are to be replaced with their corresponding value.

Listing 1 Redis database schema for SOR-Set

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

The topology is stored at key topology as a set of strings, each string representing a Redis store identified by the unique pair of cluster and shard identifiers *rc*:*rs* and *ip*:*port* giving the machine address and port number the Redis server is listening on. Each cell of the timestamp matrix, $T[rc][rs]$, is stored at key timestamp:*rc*:*rs*.

Instead of using different sets for added and removed tuples, they can be combined and stored together as *elements*. An element is created when a new value is added to the set and has the following fields: the string value e, the timestamp t, and the ids for the source replica cluster and shard where the value was added, (rc, rs). There are also equivalent fields for retaining information about removal: timestamp t' and shard (rc', rs'), which are empty in the beginning. These last fields will be populated when removing all e elements from the set. Thus, an if an element has empty rmv.t, rmv.rc, and rmv.rs fields, then it represents an *ADD* tuple, otherwise it represents an *RMV* one. 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 8 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 will be 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. As shown in the next section, this is needed for *remove*(e) and *lookup*(e) methods.

4.3 Operations Implementation

The client is implemented in Java and communication with Redis is done through the Jedis [30] library[1]. This section presents the pseudocode for main methods in the client's interface.

4.3.1 Add

Listing 2 performs the *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[2]. 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. An observation should be made regarding <u>index</u>:rc:rs: the elements are always pushed to the left (the head) of the list, thus keeping it sorted in descending order by the timestamp t. Section 4.3.4 gives an explanation for this. Procedure ADD should execute atomically and in isolation with other Redis commands on the store.

4.3.2 Remove

Listing 3 contains the pseudocode for *remove* method. 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 <u>ids</u>:e. Next, for each element stored at <u>element</u>:gid, if it is not yet expired (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 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. Procedure REMOVE should execute atomically and in isolation with other Redis commands on the store.

4.3.3 Lookup

Referring back to Specification 8, 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

[1]The client library is property of 1&1 Internet AG. Access to the source code can be allowed by request to 1&1 Internet AG, München.

[2]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 timestamp:$rc$:$rs$
    $id \leftarrow$ incr element:next.id
    hmset element:$rc.rs.id$  value $e$
                              add.t $t$
                              add.rc $rc$
                              add.rs $rs$
    expire element:$rc.rs.id$ $ttl$
    lpush index:$rc$:$rs$ $t$:$rc.rs.id$
    sadd ids:$e$ $rc.rs.id$
```

Listing 3 Redis SOR-Set: *remove*

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

Listing 4 Redis SOR-Set: *lookup*

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

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. Procedure LOOKUP should execute atomically and in isolation with other Redis commands on the store.

4.3.4 Merge

The code for last method is given in Listing 5. Here, lines 2 and 3 compute the version matrices for both the local and the remote clusters. VERSION function previously introduced in Specification 8 computes matrix $\tilde{T}$ cell by cell, by fetching timestamp:rc_i:rc_i^j, $\forall rc_i, \forall rs_i^j, j \in \{1, \ldots, |rc_i|\}$ from each store in the cluster and by selecting the minimum one (or maximum for row $rc_i^j = rc$).

Based on the local version matrix $\tilde{T}_x$, next the procedure fetches the updates from remote cluster rc_y using GetUpdates function. Remember that each store keeps an index of element ids (*ADD* and *RMV* tuples), sorted in descending order by their timestamp, i.e. the newer elements are at the beginning of the list. Since the goal is to filter all elements newer than a given $t = \tilde{T}_x[rc_i][rs_i^j]$ value, *pages* (or segments)

Listing 5 Redis SOR-Set: *merge* (part 1)

```
procedure MERGE($rc_x$, $rc_y$)
    $\tilde{T}_x \leftarrow$ VERSION($rc_x$)
    $\tilde{T}_y \leftarrow$ VERSION($rc_y$)
    $updates \leftarrow$ GETUPDATES($rc_y$, $\tilde{T}_x$)
    ADDUPDATES($rc_x$, $hash_x$, $updates$)
    UPDATETIMESTAMPS($rc_x$, $\tilde{T}_y$)
function VERSION($rc$)
    for all $rc_i$ in clusters() do
        for all $rs_i^j$ in shards($rc_i$) do
            if $rc_i = rc$ then
                $\tilde{T}[rc_i][rs_i^j] \leftarrow -\infty$
            else
                $\tilde{T}[rc_i][rs_i^j] \leftarrow +\infty$
            for all $store$ in shards($rc$) do
                $t \leftarrow store.\texttt{get}\ \underline{\text{timestamp}}{:}rc_i{:}rc_i^j$
                if $rc_i = rc$ then
                    $\tilde{T}[rc_i][rs_i^j] \leftarrow max(\tilde{T}[rc_i][rs_i^j], t)$
                else
                    $\tilde{T}[rc_i][rs_i^j] \leftarrow min(\tilde{T}[rc_i][rs_i^j], t)$
    return $\tilde{T}$
function GETUPDATES($rc$, $\tilde{T}$)
    for all $store$ in shards($rc$) do
        for all $rc_i$ in clusters() do
            for all $rs_i^j$ in shards($rc_i$) do
                $list \leftarrow store.\texttt{llen}\ \underline{\text{index}}{:}rc_i{:}rs_i^j$
                $page \leftarrow min(PAGE, list)$
                for ($start \leftarrow -list$; $start < 0$; $start \leftarrow start + page$) do
                    $tgids \leftarrow store.\texttt{lrange}\ \underline{\text{index}}{:}rc_i{:}rs_i^j\ start\ (start + page - 1)$
                    $k \leftarrow bsearch(tgids, \tilde{T}[rc_i][rs_i^j])$
                    for ($u \leftarrow 0$, $t{:}gid \leftarrow tgids[u]$; $u < k$; $u \leftarrow u + 1$) do
                        $(e, t, rc, rs, t', rc', rs')_u \leftarrow store.\texttt{hgetall}\ \underline{\text{element}}{:}gid$
                        $ttl \leftarrow store.\texttt{ttl}\ \underline{\text{element}}{:}gid$
                        $updates \leftarrow updates + ((e, t, rc, rs, t', rc', rs')_u, gid, ttl)$
                    if $k < page$ then
                        break
    return updates
```

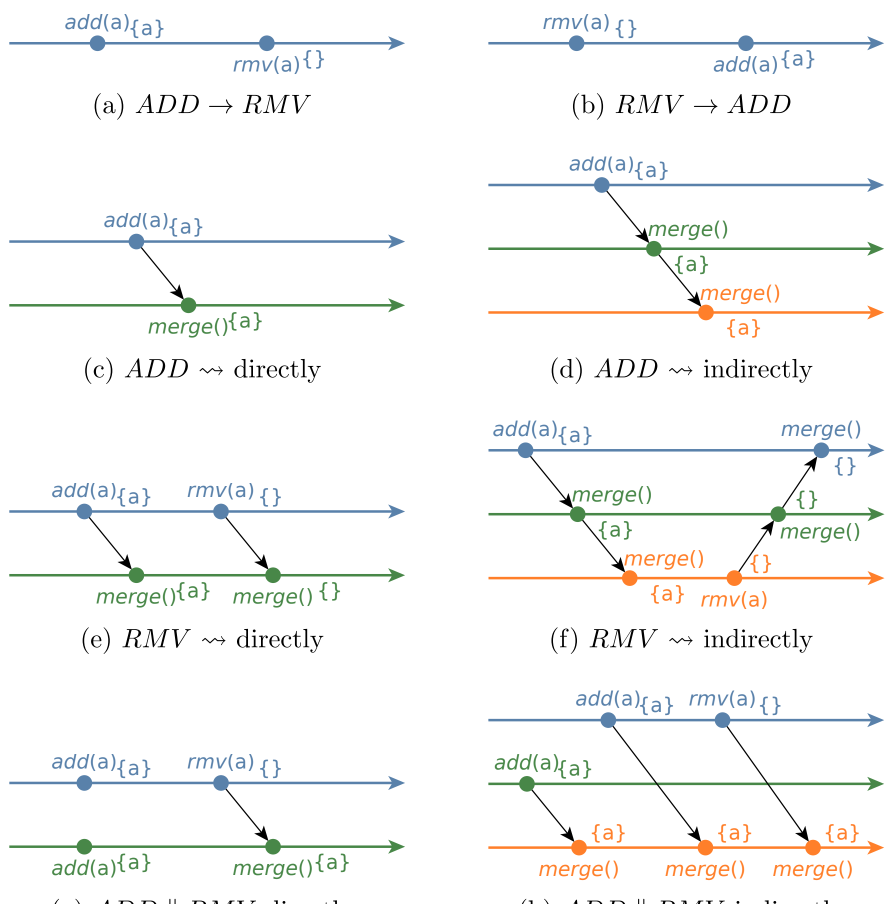
Listing 6 Redis SOR-Set: *merge* (part 2)

```
procedure AddUpdates($rc$, $hash$, $updates$)
    for all $((e, t, rc, rs, t', rc', rs')_u, gid, ttl)$ in $updates$ do
        $j \leftarrow hash(e)$
        $store \leftarrow shards(rc)[j]$
        $store$.hmset element:$gid$ $(e, t, rc, rs, t', rc', rs')_u$
        $store$.expire element:$gid$ $ttl$
        if $t' = $ null then
            $store$.lpush index:$rc$:$rs$ $t$:$gid$
        else
            $store$.lpush index:$rc'$:$rs'$ $t'$:$gid$
        $store$.sadd ids:$e$ $gid$
procedure UpdateTimestamps($rc$, $\tilde{T}$)
    for all $store$ in shards($rc$) do
        for all $rc_i$ in clusters() do
            for all $rs_i^j$ in shards($rc_i$) do
                $t \leftarrow store$.get timestamp:$rc_i$:$rs_i^j$
                $store$.set timestamp:$rc_i$:$rs_i^j$ max($t$, $\tilde{T}[rc_i][rs_i^j]$)
```

are fetched starting from the head of this list until the first value greater than t is found. All elements before this value in the list are the relevant updates. Inside each page, searching for t can be done using a binary search function because the list is already sorted. The reason why it is sorted in descending order is because Redis store should not block until fetching all pages. The store can process other requests meanwhile and, if newer element ids are added to the index, this will shift the offsets for the subsequent pages. Alternatively, the list could have been kept sorted in ascending order and fetching pages would have been done starting from the end of it. However, the complexity of one page request would be in this case proportional to the start offset of the page as stated in the Redis documentation for `lrange` command [6].

Thus, considering that p pages are fetched in total until t is found and that the page size is $page$, the complexity of GetUpdates function is $\mathcal{O}(p \times page)$. Another option would have been to use a Redis sorted set instead of a list. In this case, the complexity for each update to the index is $\mathcal{O}(log(n))$, instead of $\mathcal{O}(1)$, where n is the index size, while getting all m ids newer than t is an $\mathcal{O}(log(n)+m)$ operation. Since the page size $page$ is a heuristic that can be determined if we know the approximate number of updates between merge operations, it can be argued that just a few number of pages are needed to be fetched, and thus $\mathcal{O}(p \times page) \approx \mathcal{O}(page) < \mathcal{O}(log(n) + m)$.

Continuing with the *merge* method, *updates* contains all elements from the remote cluster, rc_y, which are missing from the local one, rc_x. In this list, elements coming from any rs_y^j store are found in the same relative order in which they were in the index of that store. Also, in conformance to the garbage collection mechanism



(a) $ADD \rightarrow RMV$ (b) $RMV \rightarrow ADD$

(c) $ADD \rightsquigarrow$ directly (d) $ADD \rightsquigarrow$ indirectly

(e) $RMV \rightsquigarrow$ directly (f) $RMV \rightsquigarrow$ indirectly

(g) $ADD \parallel RMV$ directly (h) $ADD \parallel RMV$ indirectly

Figure 4.3: Basic correctness tests for SOR-Set

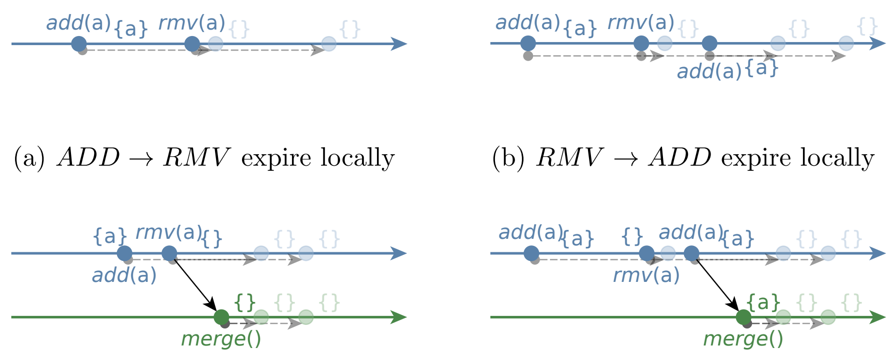
described in Section 3.4, the procedure fetches the TTLs together with the elements[3]. The next subroutine, AddUpdates, distributes these elements from *updates* to the stores in the local cluster according to $hash_x$ function. Adding an update element to the store is an operation similar to *add*, 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 $T_x^j := max(T_x^j, \tilde{T}_y), \forall j \in \{1, \ldots, |rc_x|\}$.

Procedure AddUpdates and updating the timestamps in lines 52–53 should execute atomically and in isolation with other Redis commands on that store.

4.4 Unit Tests

To prove the correctness of the SOR-Set, three categories of tests were devised: i) basic set operations, ii) garbage collection, and iii) fault tolerance. For each replica

[3]Redis command `ttl` retrieves the remaining time-to-live for the given key.



(a) *ADD* → *RMV* expire locally

(b) *RMV* → *ADD* expire locally

(c) *ADD* → *RMV* expire remotely

(d) *RMV* → *ADD* expire remotely

Figure 4.4: Garbage collection tests for SOR-Set

both single-shard and multi-shards clusters were used.

The purpose of the first set, depicted in Figure 4.3, was to test the results of update operations in different scenarios. After each update, assertions compared the boolean value returned by the $lookup(a)$ method with the expected one. The states of the replicas at each step are depicted in curly braces.

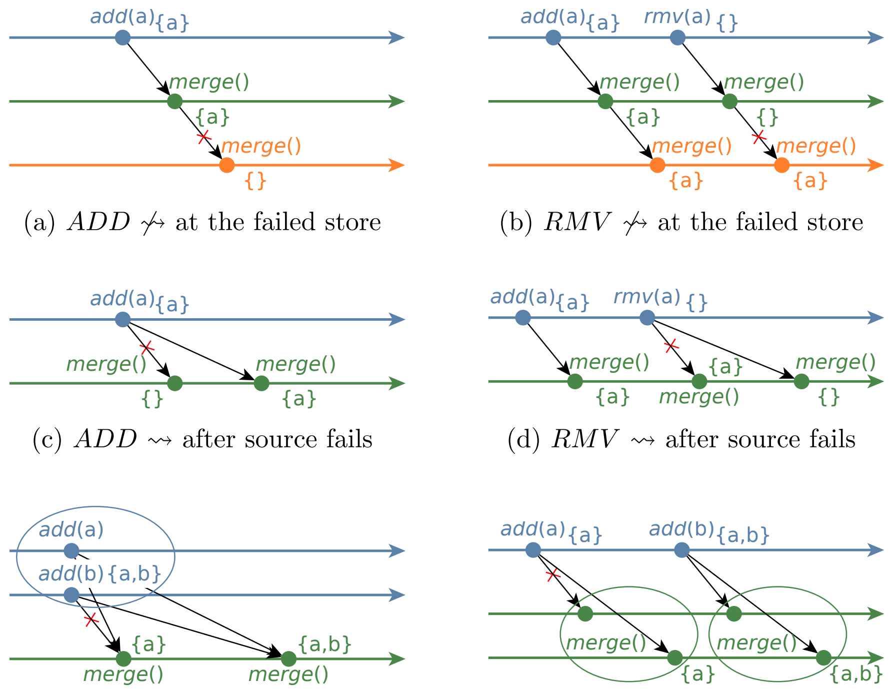
The following tests, in Figure 4.4, show that the garbage collection mechanism conforms with the tuple expiration theorem. Here, the dashed lines denote the lifetime of each tuple.

The last group of tests in Figure 4.5 proves the resilience of the SOR-Set in the presence of failures. The first two tests show that propagation of updates stops at the failed stores, while the next two show that missed updates are recovered during subsequent *merge* operations. Synchronization between replicas containing either failed stores in the remote cluster from where the updates are pulled or in the local cluster is also not affected, as shown in the last two tests.

4.5 Evaluation

This section presents the results obtained for evaluating the SOR-Set client library. The test systems were 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 was used.

The purpose of the first benchmark was to measure how the average time needed to merge two replicated sets changes as the database size increases. Test configuration included 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 was used. The client library was deployed on a third machine. The



(a) *ADD* $\not\rightsquigarrow$ at the failed store

(b) *RMV* $\not\rightsquigarrow$ at the failed store

(c) *ADD* $\rightsquigarrow$ after source fails

(d) *RMV* $\rightsquigarrow$ after source fails

(e) Store failures in remote cluster

(f) Store failures in local cluster

Figure 4.5: Fault tolerance tests for SOR-Set

methodology for measuring was: 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 Figure 4.6. Here were also included the average timings for each subroutine of the *merge* procedure described in Listing 5 relevant to these measurements: getting the pages with 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. Since the number of updates between each merge operation was constant, the timings were also relatively constant. Thus, the *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 graphic plot the average throughput for merging can be computed to 125,000 update elements per second.

The second benchmark measured the average throughput of all the basic set operations. For this, a machine with 16 Redis servers acting as a replica cluster was used. The client library was 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 Figure 4.7 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. Specifications 2, 3, and 4 show that only $\mathcal{O}(1)$ Redis

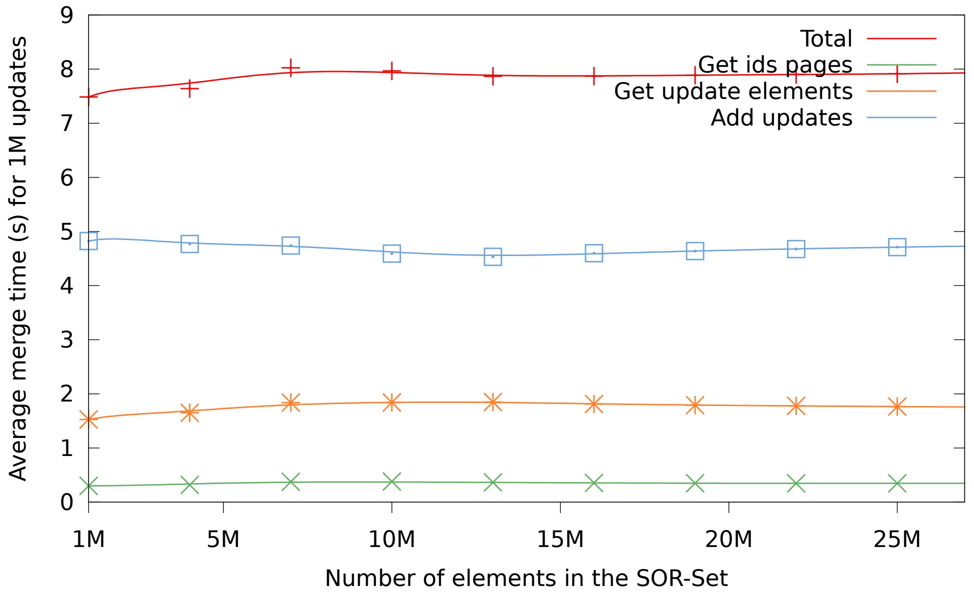


Figure 4.6: Delta-based synchronization

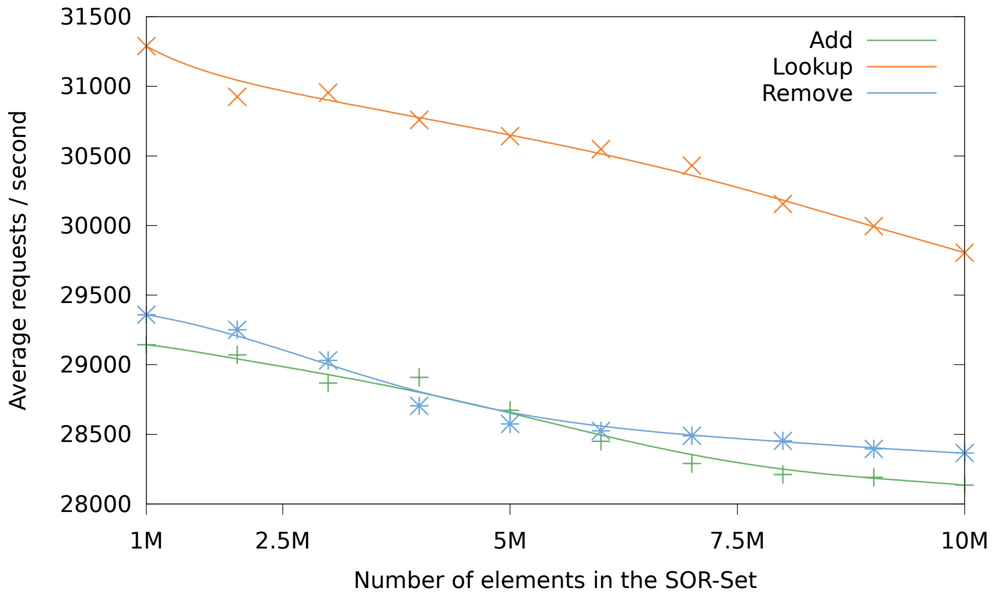


Figure 4.7: Throughput for set operations

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 $2^{8\times 32}$ space, and thus leading to a low chance of collision. Therefore, updating the indices is on average a constant operation: ids:e contains only one id and `lpush` 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.

The reason why a better throughput was obtained for *merge* has two causes. First, each of *add*, *lookup*, and *remove* is implemented using Lua scripts[4] for which Redis guarantees to execute in an atomic way. As discussed in Section 4.3, this was 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.

[4]Redis supports Lua scripts starting from version 2.6.0.

Chapter 5

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]. These features come at the cost of anomalies for some types discussed in Section 2.8, which are in general acceptable and application specific.

On the practical side, CRDTs can be deployed in various topologies since the synchronization procedure between replicas is flexible: pulling the updates can be done at any time and from any replica depending on requirements. Moreover, they can be used in conjunction with existing distribution protocols, such as gossip or anti-entropy [9, 10].

This thesis gave a theoretical background introduction on CRDTs and presented the trade-offs for each of the two styles. A number of practical examples of such types were given with a focus on the set container. The main contributions included an algorithm for efficient delta-based synchronization, an extension to the state-based set to support per-replica partitioning, and a garbage collection mechanism to alleviate the problem of unbounded database growth. To test these concepts in practice, the design and implementation of a client library together with results on evaluating its performance were given.

Future work can be done to improve the implementation details of the client library. One example is the dynamic estimation of the page size in the algorithm for the *merge* procedure from Listing 5 using heuristics such as the elements in the current page. If one can analyse this series of elements and infer a curve fitting formula for the timestamps, then a more accurate estimation on the next page size can be made to increase the probability of finding the required timestamp value.

A second improvement to the library concerns the garbage collection mechanism. Currently, only elements are expired from the database, but the indices remain untouched. One way to handle this is to run a periodic scan and to remove index entries which have no corresponding elements. Alternatively, a Redis sorted set can be used: for each update to the database, the id of the new element together with its expiration time are inserted to this set. Since the set is sorted by the expiration time, it is easy to find out which elements are expired and remove their ids from the indices.

Lastly, given the modularity of CRDTs, one can build more complex structures by composing simpler ones. Graphs, as seen in Section 2.8, can be specified using two CRDT sets, one for vertices and another for edges. Thus, SOR-Sets constitute important basic blocks in architectural design of replicated databases. Employing graph-like types in such systems helps to associate connexions between logically related elements from different sets according to specific criteria and semantics, as discussed in Section 3.1. Another example is the design of a web search engine, where a graph may be used to compute page ranks [8]. In this case, pages are stores as vertices and links between them as edges.

Bibliography

[1] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 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, 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*, SOSP '95, (New York, NY, USA), pp. 172–182, ACM, 1995.

[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*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

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

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

[8] 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*, SSS'11, (Berlin, Heidelberg), pp. 386–400, Springer-Verlag, 2011.

[9] 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*, PODC '87, (New York, NY, USA), pp. 1–12, ACM, 1987.

[10] 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*, SOSP ’97, (New York, NY, USA), pp. 288–301, ACM, 1997.

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

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

[13] 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, pp. 51–81, Feb. 1988.

[14] N. Preguiça, M. Shapiro, and C. Matheson, “Semantics-based reconciliation for collaborative and mobile environments,” in *COOPIS* (D. C. S. e. a. Robert Meersman, Zahir Tari, ed.), vol. 2888 of *Lecture notes in computer science*, (Catania, Sicily, Italie), pp. 38–55, Springer, 2003.

[15] B. G. Lindsay, “Notes on distributed databases,” IBM Research Report RJ2571(33471), IBM Research Laboratory, San Jose, NY, USA, July 1979.

[16] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[17] S. Qadeer, “Verifying sequential consistency on shared-memory multiprocessors by model checking,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, pp. 730–741, Aug. 2003.

[18] D. Mosberger, “Memory consistency models,” *SIGOPS Oper. Syst. Rev.*, vol. 27, pp. 18–26, Jan. 1993.

[19] J. J. Kistler and M. Satyanarayanan, “Disconnected operation in the coda file system,” *ACM Trans. Comput. Syst.*, vol. 10, pp. 3–25, Feb. 1992.

[20] “Riak.” `http://wiki.basho.com/Riak.html`. [A distributed key-value store].

[21] C. Baquero and F. Moura, “Specification of convergent abstract data types for autonomous mobile computing,” tech. rep., Departamento de Informática, Universidade do Minho, Oct. 1997.

[22] 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*, ICDCS ’09, (Washington, DC, USA), pp. 395–403, IEEE Computer Society, 2009.

[23] “GitHub ericmoritz repository.” `https://github.com/ericmoritz/crdt`. [A GitHub repository for testing CRDTs].

[24] “GitHub dominictarr repository.” `https://github.com/dominictarr/crdt`. [A GitHub repository for testing CRDTs].

[25] “ConcoRDanT project.” `http://concordant.lip6.fr`. [A project to study the principles of CRDTs].

[26] C. Strauch, “Nosql databases.” Lecture Selected Topics on Software-Technology Ultra-Large Scale Sites, Stuttgart Media University, 2011.

[27] G. Loukas and G. Öke, “Protection against denial of service attacks: A survey,” *The Computer Journal*, 2009.

[28] F. Mattern, “Virtual time and global states of distributed systems,” in *Parallel and Distributed Algorithms*, pp. 215–226, North-Holland, 1989.

[29] “Apache Cassandra.” `http://cassandra.apache.org`. [A distributed structured key-value store].

[30] “Jedis.” `https://github.com/xetorthio/jedis`. [Jedis – a Redis Java client].