An optimized conflict-free replicated set

Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, Sérgio Duarte

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Optimized Conflict-free Replicated Set

Annette Bieniusa, INRIA & UPMC, Paris, France
Marek Zawirski, INRIA & UPMC, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Marc Shapiro, INRIA & LIP6, Paris, France
Carlos Baquero, HASLab, INESC TEC & Universidade do Minho, Portugal
Valter Balegas, CITI, Universidade Nova de Lisboa, Portugal

Sérgio Duarte CITI, Universidade Nova de Lisboa, Portugal

N° 8083

Octobre 2012

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--8083--FR+ENG

An Optimized Conflict-free Replicated Set *

Annette Bieniusa, INRIA & UPMC, Paris, France
Marek Zawirski, INRIA & UPMC, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Marc Shapiro, INRIA & LIP6, Paris, France
Carlos Baquero, HASLab, INESC TEC & Universidade do Minho, Portugal
Valter Balegas, CITI, Universidade Nova de Lisboa, Portugal
Sérgio Duarte CITI, Universidade Nova de Lisboa, Portugal

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 8083 — Octobre 2012 — 9 pages

Abstract: Eventual consistency of replicated data supports concurrent updates, reduces latency and improves fault tolerance, but forgoes strong consistency. Accordingly, several cloud computing platforms implement eventually-consistent data types.

The set is a widespread and useful abstraction, and many replicated set designs have been proposed. We present a reasoning abstraction, *permutation equivalence*, that systematizes the characterization of the expected concurrency semantics of concurrent types. Under this framework we present one of the existing conflict-free replicated data types, Observed-Remove Set.

Furthermore, in order to decrease the size of meta-data, we propose a new optimization to avoid tombstones. This approach that can be transposed to other data types, such as maps, graphs or sequences.

Key-words: Data replication, optimistic replication, commutative operations

* This research is supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208), by ERDF, COMPETE Programme, by Google European Doctoral Fellowship in Distributed Computing received by Marek Zawirski, and FCT projects #PTDC/EIA-EIA/104022/2008 and #PTDC/EIA-EIA/108963/2008.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Optimisation d'un type de données ensemble répliqué sans conflit

Résumé : La réplication des données avec cohérence à terme permet les mises à jour concurrentes, réduit la latence, et améliore la tolérance aux fautes, mais abandonne la cohérence forte. Aussi, cette approche est utilisée dans plusieurs plateformes de nuage.

L'*ensemble* (Set) est une abstraction largement utilisée, et plusieurs modèles d'ensemble répliqués ont été proposés. Nous présentons *l'équivalence de permutation*, un principe de raisonnement qui caractérise de façon systématique la sémantique attendue d'un type de données concurrent. Ce principe nous permet d'expliquer la conception un type déjà connu, *Observed-Remove Set*.

Par ailleurs, afin de diminuer la taille des méta-données, nous proposons une nouvelle optimisation qui évite les «pierres tombales». Cette approche peut se transposer à d'autres types de données, comme les mappes, les graphes ou les séquences.

Mots-clés : Réplication des données, réplication optimiste, opérations commutatives

1 Introduction

Eventual consistency of replicated data supports concurrent updates, reduces latency and improves fault tolerance, but forgoes strong consistency (e.g., linearisability). Accordingly, several cloud computing platforms implement eventually-consistent replicated sets [3, 11]. Eventual Consistency, allows concurrent updates at different replicas, under the expectation that replicas will eventually converge [13]. However, solutions for addressing concurrent updates tend to be either limited or very complex and error-prone [7].

We follow a different approach: Strong Eventual Consistency (SEC) [9] requires a deterministic outcome for any pair of concurrent updates. Thus, different replicas can be updated in parallel, and concurrent updates are resolved locally, without requiring consensus. Some simple conditions (e.g., that concurrent updates commute with one another) are sufficient to ensure SEC. Data types that satisfy these conditions are called Conflict-Free Replicated Data Types (CRDTs). Replicas of a CRDT object can be updated without synchronization and are guaranteed to converge. This approach has been adopted in several works [15, 12, 6, 14, 11].

The set is a pervasive data type, used either directly or as a component of more complex data types, such as maps or graphs. This paper highlights the semantics of sets under eventual consistency, and introduces an optimized set implementation, *Optimized Observed Remove Set*.

2 Principle of Permutation Equivalence

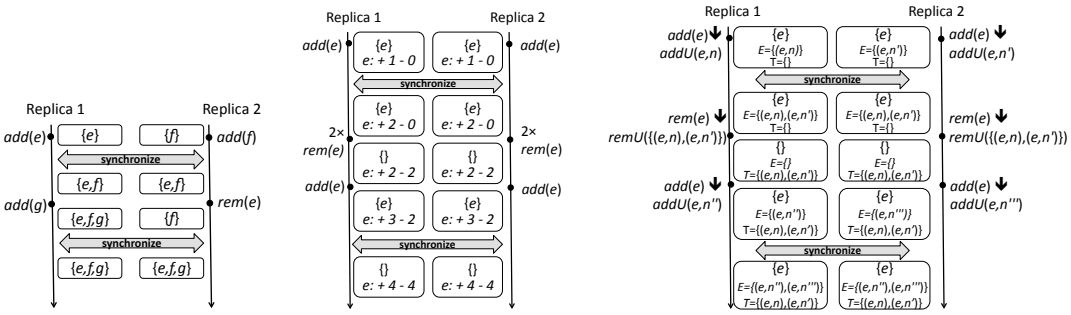
The sequential semantics of a set are well known, and are defined by individual updates, e.g., $\{\mathsf{true}\}add(e)\{e \in S\}$ (in "{pre-condition} computation {post-condition}" notation), where S denotes its abstract state. However, the semantics of concurrent modifications is left underspecified or implementation-driven.

We propose the following *Principle of Permutation Equivalence* [2] to express that concurrent behaviour conforms to the sequential specification: *"If all sequential permutations of updates lead to equivalent states, then it should also hold that concurrent executions of the updates lead to equivalent states."* It implies the following behavior, for some updates u and u':

$$\{P\}u; u'\{Q\} \wedge \{P\}u'; u\{Q'\} \wedge Q \Leftrightarrow Q' \;\Rightarrow\; \{P\}u \parallel u'\{Q\}$$

Specifically for replicated sets, the Principle of Permutation Equivalence requires that $\{e \neq f\}add(e) \parallel remove(f)\{e \in S \wedge f \notin S\}$, and similarly for operations on different elements or idempotent operations. Only the pair $add(e) \parallel remove(e)$ is unspecified by the principle, since $add(e); remove(e)$ differs from $remove(e); add(e)$. Any of the following post-conditions ensures a deterministic result:

$\{\bot_e \in S\}$	– Error mark
$\{e \in S\}$	– *add* wins
$\{e \notin S\}$	– *remove* wins
$\{add(e) >_{\mathsf{CLK}} remove(e) \Leftrightarrow e \in S\}$	– Last Writer Wins (LWW)



(a) Dynamo shopping cart (b) C-Set (c) OR-Set

Figure 1: Examples of anomalies and a correct design.

where $<_{\mathsf{CLK}}$ compares unique clocks associated with the operations. Note that not all concurrency semantics can be explained as a sequential permutation; for instance no sequential execution ever results in an error mark.

3 A Review of Existing Replicated Set Designs

In the past, several designs have been proposed for maintaining a replicated set. Most of them violate the Principle of Permutation Equivalence (Fig. 1). For instance, the Amazon Dynamo shopping cart [3] is implemented using a register supporting *read* and *write* (assignment) operations, offering the standard sequential semantics. When two *write*s occur concurrently, the next *read* returns their union. As noted by the authors themselves, in case of concurrent updates even on unrelated elements, a *remove* may be undone (Fig. 1(a)).

Sovran et al. and Asian et al. [11, 1] propose a set variant, C-Set, where for each element the associated *add* and *remove* updates are counted. The element is in the abstraction if their difference is positive. C-Set violates the Principle of Permutation Equivalence (Fig. 1(b)). When delivering the updates to both replicas as sketched, the add and remove counts are equal, i.e., e is not in the abstraction, even though the last update at each replica is $add(e)$.

4 Add-wins Replicated Sets

In Section 2 we have shown that when considering concurrent *add* and *remove* operations over the same element, one among several post-conditions can be chosen. Considering the case of an *add* wins semantics we now recall [9] the CRDT design of an Observed Remove Set, or *OR-Set*, and then introduce an optimized design that preserves the *OR-Set* behaviour and greatly improves its space complexity.

```
payload set E, set T                                   -- E: elements; T: tombstones
                                                       -- sets of pairs { (element e, unique-tag n), ... }
initial ∅, ∅
query contains (element e) : boolean b
    let b = (∃n : (e, n) ∈ E)
query elements () : set S
    let S = {e|∃n : (e, n) ∈ E}
update add (element e)
    prepare (e)
        let n = unique()                               -- unique() returns a unique tag
    effect (e, n)
        E := E ∪ {(e, n)} \ T                          -- e + unique tag
update remove (element e)
    prepare (e)                                        -- Collect pairs containing e
        let R = {(e, n)|∃n : (e, n) ∈ E}
    effect (R)                                         -- Remove pairs observed at source
        E := E \ R
        T := T ∪ R
compare (A, B) : boolean b
    let b = ((A.E ∪ A.T) ⊆ (B.E ∪ B.T)) ∧ (A.T ⊆ B.T)
merge (B)
    E := (E \ B.T) ∪ (B.E \ T)
    T := T ∪ B.T
```

Figure 2: OR-Set: Add-wins replicated set

These CRDT specifications follow a new notation with mixed state- and operation-based update propagation. Although the formalization of this mixed model, and the associated proof obligations that check compliance to CRDT requisites, is out of the scope of this report the notation is easy to infer from the standard CRDT model [9, 8, 10].

System model synopsis: We consider a single object, replicated at a given set of processes/replicas. A client of the object may invoke an operation at some replica of its choice, which is called the *source* of the operation. A query executes entirely at the source. An update applies its side effects first to the source replica, then (eventually) at all replicas, in the *downstream* for that update. To this effect, an update is modeled as an *update pair* (p, u) that includes two operations such that p is a side-effect free *prepare(-update)* operation and u is an *effect(-update)* operation; the source executes the prepare and effect atomically; downstream replicas execute only the effect u. In the mixed state- and operation-based modelling, replica state can both be changed by applying an effect operation or by merging state from another replica of the same object. The monotonic evolution of replica states is described by a compare operation, supplied with each CRDT specification.

4.1 Observed Remove Set

Figure 2 shows our specification for an add-wins replicated set CRDT. Its concurrent specification $\{P\}u_0 \parallel \ldots \parallel u_{n-1}\{Q\}$ is for each element e defined as follows:

- $\forall i, u_i = remove(e) \Rightarrow Q = (e \notin S)$
- $\exists i : u_i = add(e) \Rightarrow Q = (e \in S)$.

To implement add-wins, the idea is to distinguish different invocations of $add(e)$ by adding a hidden unique token n, and effectively store (e, n) pair. A pair (e, n) is removed by adding it to a tombstone set. An element can be always added again, because the new pair (e, n') uses always a fresh token, different from the old one, $n' \neq n$. If the same element e is both added and removed concurrently, the update-prepare of *remove* concerns only observed pairs $(e, n_1), (e, n_2), \ldots$ and not the concurrently-added unique pair (e, n'). Therefore the *add* wins by adding a new pair. We call this object an Observed Remove Set, or *OR-Set*. As illustrated in Figure 1(c), *OR-Set* is immune from the anomaly that plagues C-Set.

Space complexity: The payload size of *OR-Set* is at any moment bounded by the number of all applied add (*effect-update*) operations.

4.2 Optimized Observed Remove Set

The *OR-Set* design uses extensively unique identifiers and tombstones, as other CRDTs [6, 14, 8]. We now show how to make CRDT practical by minimizing the required meta-data.

Immediately discarding tombstones: When comparing two payloads P and P', respectively containing some element e and the other not, it is important to know if e has been recently added to P, or if it was recently removed from P'. The presented add-wins set uses tombstones to unambiguously answer this question, even when updates are delivered out of order or multiple times.

Tombstones accumulate (as a consequence of the monotonic semilattice requirement); if they cannot be discarded, memory requirements grow with the number of operations. To address this issue, Wuu's 2P-Set [15] garbage-collects tombstones that have been delivered everywhere, basically by waiting for acknowledgements from each process to every other process. This adds communication and processing overhead, and requires all processes to be correct. We devise a novel technique to eliminate tombstones without these limitations and offer conflict-free semantics at an affordable cost. We present our solution using add-wins as the example.

To recapitulate, in *OR-Set*, adding an element e creates a new unique (e, n) pair to the E part of the payload. Removing an element moves all pairs containing e observed at the source from E to T.[1] Note that adding some pair (e, n) always happens-before removing the same pair (e, n). If updates are delivered only in causal order,once, the *add* always executes before any related *remove*s, and the tombstone set T is not necessary when executing operations. However, we also need to support state-based merge, which joins two replicas possibly unrelated by happens-before. When merging two replicas in which only

[1] A practical implementation will just set a mark bit on the representation of the removed pair and will deallocate any other associated storage. Consider for instance the extension of *OR-Set* to a map: a key will have some associated value, e.g., E would contain triples $(e, n, value)$. When the key is removed, *value* can be discarded, but the corresponding (e, n) pair(s) must remain in T.

```
payload set E, vect v                    -- E: elements, set of triples (element e, timestamp c, replica i)
                                         -- v: summary (vector) of received triples
initial ∅, [0, ..., 0]
query contains (element e) : boolean b
    let b = (∃c, i : (e, c, i) ∈ E)
query elements () : set S
    let S = {e|∃c, i : (e, c, i) ∈ E}
update add (element e)
    prepare (e)
        let r = myID()                   -- r = source replica
        let c = v[r] + 1
    effect (e, c, r)
        pre causal delivery
        if c > v[r] then
            let O = {(e, c', r) ∈ E|c' < c}
            v[r] := c
            E := E ∪ {(e, c, r)} \ O
update remove (element e)
    prepare (e)                          -- Collect all unique triples containing e
        let R = {(e, c, i) ∈ E}
    effect (R)                           -- Remove triples observed at source
        pre causal delivery
        E := E \ R
compare (A, B) : boolean b
    let R = {(c, i)|0 < c ≤ A.v[i] ∧ ∄e : (e, c, i) ∈ A.E}
    let R' = {(c, i)|0 < c ≤ B.v[i] ∧ ∄e : (e, c, i) ∈ B.E}
    let b = A.v ≤ B.v ∧ R ⊆ R'
merge (B)
    let M = (E ∩ B.E)
    let M' = {(e, c, i) ∈ E \ B.E|c > B.v[i]}
    let M'' = {(e, c, i) ∈ B.E \ E|c > v[i]}
    let U = M ∪ M' ∪ M''
    let O = {(e, c, i) ∈ U|∃(e, c', i) ∈ U : c < c'}
    E := U \ O
    v := [max(v[0], B.v[0]), ..., max(v[n], B.v[n])]
```

Figure 3: Optimized OR-Set (Opt-OR-Set).

one replica has some pair (e, n), we need to know if the pair has been added to the replica that contains it or if it was removed in the other replica.

We leverage these observations to propose a novel *remove* algorithm that discards a removed pair immediately and works safely with merge. It compactly records happens-before information to summarizes removed elements. Figure 3 presents *Optimized OR-Set* (Opt-OR-Set) based on this approach.

Each replica i maintains a vector v [5] to summarize the unique identifiers it has already observed. Entry $v[j] = n$ at replica i indicates that this replica has observed n successive identifiers generated at j: $(1, j), (2, j), \ldots, (n, j)$. Replica i maintains its local counter as the i-th entry in the vector $v[i]$, initially 0. A replica generates new unique identifiers (c, i)

by incrementing its local counter. Note that to summarize successive identifiers in a vector, OptORSet requires causal delivery of updates.[2]

When *add* is invoked, the source associates it with a unique identifier made of the next local counter value and source replica identifier. When the *add* is delivered to a downstream replica, it should have an effect only if it has not been previously delivered; for this, it checks if the unique identifier is incorporated in the downstream replica's vector. When merge-ing payloads, an element should be in the merged state only if: either it is in both payloads (set M in Figure 3), or it is in the local payload and not recently removed from the remote one (set M') or vice-versa (M'') - an element has been removed if it is not in the payload but its identifier is reflected in the replica's vector.

This approach can be generalized to any CRDT where elements are added and removed, e.g., a sequence [6, 14] or a graph [10].

Coalescing repeated adds: Another source of memory growth in the original *OR-Set* is due to the elements added several times. Similarly to tombstones, they pollute the state with unique identifiers for every *add*. We observe that for every combination of element and source replica, it is enough to keep the identifier of the latest *add*, which subsumes previously added elements. The OptORSet specification leverages this observation in *add* and merge definitions, by discarding unnecessary identifiers (set O).

Space complexity: The payload size of OptORSet set is bounded by $O(|elements|n+n)$ at any moment, where n is the number of processes in the systems and $|elements|$ is the number of elements present in the set. The first component corresponds to the maximum number of timestamps in set E and the second captures the size of the vector v. In the common case, where the number of processes repeatedly invoking *add*s can be considered a constant, the payload size is $O(|elements| + n)$.

5 Conclusions

Conflict-Free Replicated Data Types (CRDTs) allow a system to maintain multiple replicas of data that are updated without requiring synchronization while guaranteeing Strong Eventual Consistency. This allows, for example, a cloud infrastructure to maintain replicas of data in data centers spread over large geographical distance and still provide low access latency by choosing the closest, to client, data center.

In this paper we reviewed existing replicated set designs and contrasted then with the CRDT *OR-Set* design, under the principle of permutation equivalence. Having in mind that the base *OR-Set* favored simplicity at the expense of scalability, we introduced a new optimized design, *Optimized OR-Set*, that greatly improves its scalability and should favor efficient implementations of sets and other CRDTs that share the *OR-Set* design techniques.

[2] It is easy to extend this solution for updates delivered out of happens-before order by using instead a version vector with exceptions [4].

References

[1] Khaled Aslan, Pascal Molli, Hala Skaf-Molli, and Stéphane Weiss. C-Set: a commutative replicated data type for semantic stores. In *RED: Fourth International Workshop on REsource Discovery*, Heraklion, Greece, 2011.

[2] Annette Bieniusa, Marek Zawirsky, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief annoucement: Semantics of eventually consistent replicated sets. In *Proceedings of the 26th international conference on Distributed Computing*, DISC'12, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pages 205–220, Stevenson, Washington, USA, October 2007. Assoc. for Computing Machinery.

[4] Dahlia Malkhi and Douglas B. Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):209–219, 2007.

[5] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9:240–247, May 1983.

[6] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Leţia. A commutative replicated data type for cooperative editing. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 395–403, Montréal, Canada, June 2009.

[7] Yasushi Saito and Marc Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42–81, March 2005.

[8] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche 7506, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, January 2011.

[9] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag GmbH.

[10] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (104):67–88, June 2011.

[11] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Computing Machinery.

[12] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *Trans. on Computer Systems*, 4(2):180–209, June 1979.

[13] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008.

[14] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on P2P networks. *IEEE Trans. on Parallel and Dist. Sys. (TPDS)*, 21:1162–1174, 2010.

[15] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 233–242, Vancouver, BC, Canada, August 1984.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399