# Position paper: CRDTs for large-scale incremental processing[*]

Nuno Preguiça
U. Nova de Lisboa and INRIA

Marc Shapiro
INRIA and LIP6

Marek Zawirski
U. Pierre et Marie Curie and INRIA

Revision 82 (2011-06-17)

## Abstract

We propose a new approach for building distributed large-scale applications such as a social network or a search engine. It is based on high-level, conflict-free abstract data types interconnected with a dataflow-style notification network. This approach allows to replicate objects widely, close to the edge of the network, and to push small update notifications. Computation is expressed in atomic processing steps which allow to work on consistent snapshots of system state and execute updates in transaction-like manner.

## 1 Introduction

We propose a new, highly-scalable, low-latency approach supporting data-oriented applications for cloud or peer-to-peer settings.

The CAP theorem predicts a trade-off between constency, availability and fault tolerance [5]. Contemporary noSQL stores, such as DHTs [4], memcached or Bigtable [3] are very scalable, but provide very weak consistency guarantees. Some recent stores support system-wide transactions, e.g., Megastore [1]. Percolator [9] shows that a class of computations can be structured instead as small, incremental transactions interconnected by a notification network. This improves responsiveness and scalability; as an added bonus, the straight-line transactional code is simpler. Transactions, however, remain costly, requiring high-latency synchronisation.

We argue that there is fruitful middle ground to be explored. A fundamental issue is that current stores offer only a low-level read-write interface; when manipulating high-level information this is awkward[1] and forces the use of transactions even for simple, incremental operations such as adding or deleting an element in a set. We suggest instead that the infrastructure support the high-level incremental update operations of abstract data types. We carefully design *Conflict-free Replicated Data Types* (CRDTs), i.e., in which any replica can be updated independently, with no synchronisation. Every CRDT replica is always responsive, even when other replicas have crashed or when the network is partitioned; and an update costs only a procedure call to the closest replica.

Non-trivial CRDTs have been presented before [7, 10, 11]. This paper's contribution is to explore how one might build a large data-intensive application using CRDTs. We consider here the example of a search engine, but we believe the same approach can be used in many other applications, e.g., a social network or for stream processing in a sensor network. We consider specifically the design of replicated, sharded, and incrementally-maintained data types for large-scale storage and analysis of a dynamic network; how to compose such objects; how to take consistent snapshots and to implement atomic processing steps over CRDTs.

## 2 CRDTs

In Eventual Consistency (EC), if updates cease, replicas should eventually converge to a correct value. EC however allows tentative and conflicting updates, which in the general case requires consensus to resolve the conflict, and which causes replicas to roll back [2, 12]. To avoid these bottlenecks, we require *Strong Eventual Consistency* (SEC) defined as follows:

**Eventual delivery:** An update delivered at some correct (i.e., non-crashed) replica is eventually delivered to all correct replicas.

**Strong Convergence:** Correct replicas that have delivered the same updates have equivalent state.

**Termination:** All method executions terminate.

[*]This research is supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208). Marek Zawirski is supported in part by a Google Europe Fellowship in Distributed Computing.

[1] For instance, see the discussion on storing sets or lists in memcached [8], or the Amazon Shopping List anomalies [4].

We demonstrate elsewhere [11] two sufficient conditions for ensuring SEC. In an operation-based model, all concurrent operations must commute. In a state-based model, the successive states must form a partial order, updates must always advance in the partial order, and merging states must compute a least-upper-bound. These two conditions are equivalent in a strong sense, i.e., there is a systematic (but inefficient) transformation from a state-based to the corresponding operation-based implementation, and vice-versa. Typically, the state-based approach is simpler to reason about, and (hand-implemented) operation-based objects are more efficient. An object type that fulfills these conditions is called a Conflict-Free Replicated Data Type (CRDT).

Any subset of replicas of a CRDT that can communicate eventually converge, independently of the fate of the remaining replicas; they only need to deliver messages. Therefore a CRDT object with $n$ replicas tolerates up to $n-1$ simultaneous crashes. In this sense, a CRDT addresses the CAP problem [5], as it ensures at the same time (strong eventual) consistency, availabilty and fault tolerance. Remarkably, it does not require consensus. Any CRDT can be widely replicated and scales almost without limits.

We have designed a number of non-trivial CRDTs. At LADIS 2009 we presented Treedoc, a efficient CRDT maintaining a sequence of elements (very much like a linked list), designed for concurrent editing [7]. We have also designed counters, several variants of sets, maps, sequences, and graphs [11].

In summary, CRDTs can provide: *(i)* A high-level, incremental update interface, such as adding and removing elements in a set or map, or vertices and arcs in a graph. *(ii)* Replicating data widely, for responsiveness and availability. *(iii)* Tolerating network disruption and unavailable or crashed replicas. *(iv)* Low-cost synchronisation. *(v)* Weak but well-defined consistency guarantees.

## 3 Designing scalable abstract data types

### 3.1 Observed-Remove Set

**Semantics and specification** Sets are a widely-used data type. Our set should follow the usual sequential specification of $add$- and $remove$-element operations, but we also require concurrent operations to commute. Several alternative semantics can be considered, in particular with respect to concurrently adding and removing

```
payload set $A$, set $R$, vectorTimestamp $V$
            -- $A$: added-set of (elem $e$, timestamp $t$, replica $r$)
        -- $R$: removed-set of ($e$, $t$, $r$, timestamp $t'$, replica $r'$)
                        -- $V$: timestamp vector for snapshots
    initial $\varnothing, \varnothing, [0, \ldots, 0]$
update $add$ ($e$)
    let $g = myID()$                          -- Make each $add$ unique
    let $t = V[g] + 1$                        -- Timestamp for snapshot
    $A := A \cup \{(e, t, g)\}; V[g] := t$
update $remove$ ($e$)
    let $g' = myID()$
    let $t' = V[g] + 1$                       -- Timestamp for Snapshot
    let $R' = \{(e, t, r, t', g') | (e, t, r) \in A\}$
            -- Remove all observed unique representatives of $e$
    $R := R \cup R'; V[g] := t$
query $read$ (vectorTimestamp $T$) : set $S$
                        -- Return content at vector time $T$
    let $S = \{e | \exists (e, t, r) \in A : T[r] \geq t$
                        $\wedge \nexists (e, t, r, t', r') \in R : T[r'] \leq t'\}$
        -- Filter duplicates and consider state at vector time $T$
compare $(X, Y)$ : boolean $b$                -- Partial order over states
    $X.A \subseteq Y.A \wedge X.R \subseteq Y.R$
merge $(X, Y)$ : payload $Z$
            -- Merge states = compute Least Upper Bound
    $\forall i : Z.V[i] := max(X.V[i], Y.V[i])$
    $Z.A := X.A \cup Y.A$
    $Z.R = X.R \cup Y.R$
```

Figure 1: OR-set with snapshots (state-based)

the same element $add(e) \parallel remove(e)$: *(i)* Give precedence to the constructive operation, $add(e)$: the final state includes $e$. *(ii)* Give precedence to $remove$: $e$ is not in the final state. *(iii)* Last Writer Wins: give precedence to update with the highest timestamp. *(iv)* Replace $e$ with an error marker $\perp_e$. Any of the above satisfies commutativity (and other solutions are possible); the right option depends on the application. For the example in Section 4, option *i* is the most appropriate. Our specification in Figure 1 shows how to do this. Our specification also supports accessing a consistent snapshot, even if the set is updated at a high rate.

We now explain our specification intuitively. The state of a replica ("payload") consists of a set of added elements, a set of removed elements (also called tombstones), and a timestamp vector. Our internal representation makes every $add$ unique by associating a timestamp. Duplicates are filtered through the $read$ interface. To remove element $e$, all the unique representatives of $e$ that can be observed by the remover are added to the removed-set, hence the name Observed-Remove Set or OR-Set. Tombstones are themselves augmented with a unique remove timestamp. Timestamps will be used for taking snapshots, as explained later.[2]

**Snapshots and sharding** The OR-Set's timestamp vector keeps track of the current version; the unique identifier of an $add$ or $remove$ is the local entry of the vector. As the $read$ operation takes a timestamp vector as an argument, this suffices to support reading a consistent snapshot across multiple objects. The set includes all elements that have been added before the $read$ timestamp ($(e,t,r) \in A : T[r] \geq t$), and either not removed, or removed after the timestamp ($\nexists(e,t,r,t',r') \in R : T[r'] \leq t'$).

Assuming some well-known, stable hash function, a large OR-set can be trivially sharded by distributing the elements across multiple OR-Sets by selecting one using the hash of the element. This allows to spread OR-Set shards and replicas across different locations. Both updates and lookups incur only the cost of a hash and an access to a single replica. The only issue is that the timestamp vector must not be internal to a particular shard but must be shared across all shards, in order to be able to take a consistent snapshot across them.

## 3.2 Directed Graph CRDT

Many applications require a more complex data type, such as a Directed Graph to represent the structure of web pages or of a social network. Let us focus on the requirements of a hypothetical search engine, the example of Section 4.

**Concurrency alternatives** A directed graph is defined as a pair of sets $(V, A)$, called vertices and arcs respectively, respecting invariant $A \subseteq V \times V$. This invariant must be maintained when adding and removing vertices and arcs; therefore, such operations are not independent. This is easy in a sequential setting, but not so in the concurrent case without synchronisation. Consider for instance concurrent $addArc(v, v') \parallel removeVertex(v)$; several alternatives are reasonable: *(i)* Give precedence to $removeVertex(v)$, removing all arcs to or from $v$ as a side effect. This is easy implemented by hiding any arc that includes a removed vertex. *(ii)* Give precedence to $addArc(v, v')$: if either $v$ or $v'$ has been removed, it is restored. *(iii)* Delay $removeVertex(v)$ until all concurrent $addArc$ operations have executed. This requires global synchronisation, which violates the goals of asynchrony and fault tolerance. There is no perfect choice. As will appear clearly later, option *(i)* is the most appropriate for our example application.

**Specification** Our implementation of the Graph CRDT basically combines two OR-sets, one for vertices and one for arcs. Giving precedence to $removeVertex$, in the case of concurrent $addArc$/$removeVertex$, is implemented by maintaining a superset of the actual arcs, and hiding those whose tail or head vertex does not exist.

This design is consistent with the semantics of webpage crawling and processing, since a URL may point to a page that does not exist. When a crawler finds a new page, it invokes the corresponding $addVertex$. It processes a page comparing it with its previous version, if any, and invokes $addArc$ or $removeArc$ for every new or removed URL. The URLs of the linked pages are added to the set to be crawled. Note that $addArc$ must work even if the page at the head of the arc has not yet been found (it might not even exist), but such an arc is not functional; looking it up through the Graph interface returns false. Similarly, if a vertex has been removed, all arcs incident to it immediately disappear, without further processing. When a URL points to a non-existent page, and the page is later found, the corresponding arc immediately becomes functional.

**Snapshots and sharding** A large vertex set may be sharded by applying a static hash function, as explained earlier for OR-Set. For crawling, an arc will be stored in the same shard as its tail; in this way, when crawling a page, all $addVertex$ and $addArc$ and $removeArc$ will be processed in the same shard. For an inverted graph (such as an index) instead, it is more appropriate to store an arc in the shard of its head.

Snapshots are implemented in the same way as for OR-Set, with the caveat that the timestamp vector must be shared across shards and between the vertex and arc sets. This will ensure that a computation that accesses different parts of the graph can observe a consistent snapshot.

# 4 Putting it all together

Let us now consider a specific example, building a (hypothetical) web search engine from CRDT components.

[2] The abstract specification of Figure 1 is written to be formal and unambiguous, not necessarily efficient. Instead of maintaining a separate tombstone, an actual implementation would just mark the element as logically removed, and it would reclaim obsolete tombstones. We ignore these issues here for lack of space.

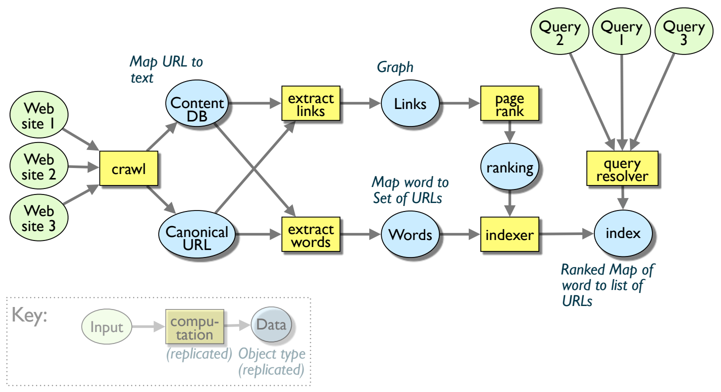

Figure 2: Percolator-like hypothetical search engine architecture

At one end, the system continuously crawls web pages. A complex computation summarises the content of the pages to produce a pre-computed index, which, at the other end, a query resolver uses to answer user queries.

The classical approach would compute the index as a single a transaction, ensuring that content remains stable and consistent. This is clearly slow and not scalable. Instead, Percolator computes the index incrementally, decomposing the computation into a data flow of small, independent transactions; this dramatically improves responsiveness [9]. Inconsistencies between successive transactions are considered minor. Concurrent write transactions may cause aborts. A new or modified web page triggers a flow of incremental updates to successive objects, eventually reaching the index.

Our own approach is illustrated in Figure 2. Objects are interconnected by a Percolator-style notification network. Each computation step is data-intensive and data-parallel, and incrementally receives updates from an input object, transforming them into updates on some output object. Objects are replicated and sharded.

Let us discuss consistency and isolation requirements. Percolator shows that snapshot isolation (SI) is sufficient for this application.[3] We argue that even SI is overkill for most of these tasks, and that the same operations can be executed accessing a consistent snapshot across multiple CRDTs and executing an atomic set of high-level CRDT operations. Consider the processing of a crawled page: it stores the new content, and possibly adds it to a registry of duplicates. Percolator performs this processing with a SI transaction per document [9]. If concurrent transactions happen to process pages with the same content, only one can register it in the duplicates table; the other aborts. Even if there is no conflict, this processing pays the high latency cost of transactional synchronization.

[3] In SI, a transaction reads a consistent snapshot of the database. A read transaction always commits. An update transaction aborts only if it writes to data that was modified since its snapshot.

Accessing a consistent snapshot is important, when processing information from multiple objects or even multiple shards. However, aborting on write conflicts could be replaced by a simple CRDT rule, such as last-writer-wins. Specifically, we would implement this processing as a set of *add* invocations to two CRDT Map objects, *contents* and *duplicates*, each of which would use a Last Writer Wins rule [6, 11] for handling updates to the same key. To ensure consistent writes, both objects use the same LWW timestamp (passed as an extra argument).

Furthermore we assume that computation is performed within atomic processing steps: input objects are always accessed in a consistent snapshot, and updates to output objects provide all-or-nothing semantics. Only all or none of updates from a processing step can be visible in a later consistent snapshot. Note, that we can ensure these guarantees using system-wide timestamp vectors, as sketched in Section 3.

Other CRDTs can be used to implement some of the other steps, e.g., building a graph of page links (the Graph from Section 3.2), clustering pages (Map of OR-Sets), creating index (e.g. Map of word to list of URLs), etc. These objects can be replicated widely, close to the network edge.

However, in general, there may remain some steps that require SI atomicity or even serializability. For these, we still have the option of implementing write-write atomicity in the usual manner. Objects involved in such transactions will have small numbers of replicas, located such that two-phase commit or atomic-multicast protocols are feasible.

## 5 Conclusions

This paper argues for the existence of a middle ground to be explored between data storage systems providing eventual consistency and those providing transactions with strong consistency guarantees. The key idea is to build a storage system with high-level abstract data types that support concurrent updates with meaningful merge functions - to achieve this, we have developed several *Conflict-free Replicated Data Types*.

With such data types, most data processing can be executed incrementally and in parallel with no synchronization. However, it is often important to access a consistent snapshot of the data. We show how to support such feature in CRDTs. We are currently building a system based on the proposed idea.

## References

[1] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Biennial Conf. on Innovative DataSystems Research (CIDR)*, pages 229–240, Asilomar, CA, USA, January 2011.

[2] Lamia Benmouffok, Jean-Michel Busca, Joan Manuel Marquès, Marc Shapiro, Pierre Sutra, and Georgios Tsoukalas. Telex: A semantic platform for cooperative application development. In *Conf. Française sur les Systèmes d'Exploitation (CFSE)*, Toulouse, France, September 2009.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[4] 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.

[5] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[6] Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, January 1976.

[7] Mihai Leţia, Nuno Preguiça, and Marc Shapiro. CRDTs: Consistency without concurrency control. In *SOSP W. on Large Scale Distributed Systems and Middleware (LADIS)*, volume 44 of *Operating Systems Review*, pages 29–34, Big Sky, MT, USA, October 2009. ACM SIG on Operating Systems (SIGOPS), Assoc. for Comp. Machinery.

[8] memcached wiki. Storing sets or lists. http://code.google.com/p/memcached/wiki/NewProgrammingTricks#Storing_sets_or_lists, Viewed 2011-06-16.

[9] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 252–264, Berkeley, CA, USA, 2010. USENIX Association.

[10] 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.

[11] 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.

[12] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th Symp. on Op. Sys. Principles (SOSP)*, pages 172–182, Copper Mountain, CO, USA, December 1995. ACM SIGOPS, ACM Press.