Replicated Data Types

Marc Shapiro

Conflict-Free Replicated Data Types (basic entry)

Marc Shapiro

Sorbonne-Universités-UPMC-LIP6 & Inria Paris
http://lip6.fr/Marc.Shapiro/

16 May 2016

1 Synonyms

Conflict-Free Replicated Data Types (CRDTs).
Commutative Replicated Data Types (CmRDTs).
Convergent Replicated Data Types (CvRDTs).
Replicated Abstract Data Types (RADTs).
Replicated Data Types (RDTs).

2 Definition

Conflict-Free Replicated Data Types (CRDTs) were invented to encapsulate and hide the complexity of managing Eventual Consistency. A CRDT is an abstract data type that implements some familiar object, such as a counter, a set or a sequence. Internally, a CRDT is replicated, to provide reliability, availability and responsiveness. Encapsulation hides the details of replication and conflict resolution.

In a sequential execution, the CRDT behaves like its sequential counterpart. Thus, a CRDT is reusable by programmers without detailed knowledge of its implementation. Furthermore, a CRDT supports concurrent updates, and encapsulates some strategy that provably ensures that replicas of the CRDT will converge despite this concurrency. Concurrent updates are never conflicting.

3 Historical background

There is precursor work on specific CRDTs, before the concept was formally identified as an independent abstraction. Johnson and Thomas [11] proposed the so-called Last-Writer-Wins (LWW) or Greatest-Timestamp-Wins approach for a replicated register,

i.e., an untyped memory that an update completely overwrites. Wuu and Bernstein [19] studied more complex data types, the log and dictionary (a.k.a. map or Key-Value Store). The whole area of Operational Transformation (OT) studied replicated strings or sequences, intended for concurrent editing applications [18]. Baquero and Moura [3] identified some convergence conditions for data types used in mobile computing. The Dynamo system is based on a multi-value register construct [9]. Related topics include replicated file systems and version control systems.

The concept of CRDTs was identified by Preguiça et al. [13], formalised by Shapiro et al. [16] and Shapiro et al. [17], and studied systematically in Shapiro et al. [15]. A similar concept, called Replicated Abstract Data Types (RADTs), was proposed independently by Roh et al. [14]. These works consider symmetric replicas, in which concurrent updates must be commutative and associative. In related work, Burckhardt et al. [7] consider so-called Cloud Types with asymmetric main and secondary branches, thus relaxing the commutativity requirement.

The distinction between *operation-* and *state-based* CRDTs was established by Shapiro et al. [16]. Burckhardt et al. [8] established lower-bound and optimality results for some representative state-based CRDTs. *Delta-CRDTs* were proposed to decrease the footprint of state-based CRDTs while keeping most of their advantages [1]. *Pure operation-based* CRDTs leverage causal-order delivery to streamline the design and implementation of operation-based CRDTs [4].

4 Foundations

4.1 Encapsulating replication and concurrency

In a distributed system, shared data is often replicated to improve the availability and latency of *reads*. However, requiring strong consistency between replicas will actually degrade the availability and latency of *writes*. According to the CAP Theorem, to improve write availability and latency requires to relax the consistency requirement: a replica should accept updates without synchronising with other replicas, and propagate them in the background.

If multiple replicas accept updates (a so-called "multi-master" system), inevitably, there will be concurrent updates to separate replicas. Managing and reconciling conflicting concurrent updates, in order to ensure Eventual Consistency, is a major issue of such systems.

CRDTs were invented to resolve this issue, by encapsulating a familiar object abstraction with a mathematically-sound conflict resolution protocol.

4.2 CRDT behaviour

A CRDT supports the interface of the corresponding abstraction. Thus, a *register* CRDT will support mutation methods such as *read* and *write* methods; a *counter* supports *increment*, *decrement* and *value* methods; a set methods to *add*, *remove* and *query* elements, and so on.

A number of CRDT types have been proposed in the literature. The most basic ones are the LWW Register [11] and the Multi-Value Register [9]. A widely-studied CRDT is the sequence or list, used in particular for cooperative editing [13, 14]. Other common CRDTs include counters [15], sets [15] and maps [14].

Consider for instance a set data type, supporting operations to add and remove elements, ignoring duplicates. An archetypical CRDT set is the so-called *Observed-Remove Set* (OR-Set). In any sequential execution, it behaves exactly like a sequential set. Concurrently adding and removing different elements e and f, or adding the same element e twice, or removing the same element e twice, commute per the sequential specification. However, to ensure commutativity of two updates that concurrently add and remove the same element e, the OR-Set makes the "add win," i.e., any replica that observes both operations concludes that e is a member of the set. To do this, the implementation of the remove operation effectively cancels out those add operations that it previously observed, and only those. We return to this example later in this entry.

4.3 Implementation approaches and requirements

A CRDT is typically designed to behave like its sequential counterpart in any sequential execution. Furthermore, a CRDT is replicated, supports *concurrent updates* for availability, and encapsulates some strategy to *merge* concurrent updates and ensure that its replicas eventually converge. One such strategy is the "Last-Writer-Wins" approach [11] that uses timestamps to totally order updates and discard all but the highest-timestamped one. Another is to record concurrent updates side-by-side, so that the application can deal with them later, as in the Multi-Value Register of Dynamo [9] and in many filesystems or version control systems.

The literature distinguishes two implementation strategies for CRDTs. In the *state-based* approach, a mutation method changes only the state of the origin replica. Periodically, a replica sends its full state to some other. The receiver *merges* the received state into its own. A state-based CRDT manages its state space as a join-semilattice, where every mutator method is an inflation, and the *merge* method computes the join (a.k.a. least-upper-bound) of the states to be merged [3, 16]. Semi-lattice join is associative, commutative and idempotent. The first two properties ensure that all replicas converge deterministically to the same outcome. The latter ensures that the system tolerates duplicated merges. As long as replicas communicate their state sufficiently often, and the communication graph is connected, replicas eventually converge and each object's history is causally consistent.

The *operation-based* approach consists of sending updates rather than states. A mutator method consists of two steps. The *generator* step reads the state of the origin replica and generates an *effector*, a state transformation that is sent and eventually applied to all replicas in the second step [10, 12, 16].[1] Concurrent effectors must commute with one another since they may be received in any order. Associativity is not required but, if available, enables batching multiple effectors into a single one. The operation-based approach requires that the underlying communication layer deliver updates to the object in causal order, and never deliver the same (non-idempotent) update twice.

The state-based approach is generally considered less efficient (state may be very large) but more elegant and simpler to understand. It makes very few assumptions about the underlying network; for instance, the number and identity of replicas may be unknown and variable. Conversely, the operation-based approach appears more efficient but is more complex to implement and requires a more elaborate communication layer.

While CRDTs replicas are guaranteed to eventually converge, this may be insufficient for application correctness. Many applications also require Causal Consistency to avoid ordering anomalies across objects. Furthermore, maintaining the integrity of structural invariants may require synchronisation to disallow certain concurrent updates [2, 10] (see also Multi Datacenter Consistency).

4.4 Example: OR-Set

The following pseudocode, in the style of Shapiro et al. [16], illustrates a state-based implementation of an OR-Set [15]. The local variables of a replica are a set E of added elements, and a set T of removed elements or tombstones. Adding an element e puts it into E along with a unique tag. The tag remains internal to the implementation and is not visible through the interface. Removing an element e moves all pairs of the form $(e, \_)$ from S into T. An element e is contained in the set if there exists a pair of the form $(e, \_)$ in E. Merging two states retains the element pairs that are contained in both states, and makes tombstones of element pairs that are tombstones in either state.

```
                                          -- State-based OR-Set specification, with tombstones
variables 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)
update add (element e)
    let n = unique()                      -- unique() returns a unique tag
    E := E ∪ {(e, n)}                     -- e + unique tag
update remove (element e)
    let R = {(e, n)|∃n : (e, n) ∈ E}      -- Collect all unique pairs containing e
```

[1] This vocabulary is not standardised. Other names for the generator phase are upstream, prepare, or prepare-update. Alternative names for the effector phase are downstream, effect, effect-update or shadow operation.

```
    E := E \ R
    T := T ∪ R                                   -- Make pairs observed at origin into tombstone
merge (B)
    E := (E \ B.T) ∪ (B.E \ T)
    T := T ∪ B.T
```

Unfortunately the memory usage of this specification grows, without bound, with every *add* and *remove* operation. However, observe that adding an element pair necessarily happens-before removing the same pair. Leveraging this observation, Bieniusa et al. [6] propose an implementation whose size is bounded by the number of currently-contained elements; since the state-based approach does not assume any particular delivery order, it is somewhat complex. As the operation-based approach already assumes causal-order delivery, avoiding tombstones is straightforward, as shown next [17]. A replica maintains a set of contained element-pairs E. Adding an element e creates the corresponding pair, and removing an element e simply removes all pairs of the form $(e, \_)$ observed at the origin replica.

```
                                   -- Operation-based Observed-Remove Set, without tombstones
variables set E                    -- set of pairs { (element e, unique-tag u), ... }
    initial ∅
query contains (element e) : boolean b
    let b = (∃u : (e, u) ∈ E)
update add (element e)
    generator (e)
        let u = unique()           -- unique() returns a unique value
    effector (e, u)
        E := E ∪ {(e, u)}          -- e + unique tag
update remove (element e)
    generator (e)
        let R = {(e, u)|∃u : (e, u) ∈ E}
                                   -- Generator: Collect all unique pairs containing e
    effector (R)
        E := E \ R
                                   -- Effector: remove pairs observed at source
```

5 Usage

Several implementations of CRDTs have been reported, in languages such as C++, Clojure, Erlang, Go, Java, Python, Ruby, and Scala.

The Riak NoSQL database, as of Version 2.0, implements a number of replicated data types, including flags, registers, counters, sets and maps [5]. Bet365, a large online betting company, which manages 2.5 million simultaneous users with Riak OR-Sets. League of Legends, an online multiplayer game, implements online chat for 70 million users with Riak sets. TomTom extend Riak's CRDTs to share navigation data between a user's different devices. SoundCloud uses a Go implementation on top of the Redis database to store time-series information.

6 Cross References

CAP Theorem.
Causal Consistency.
Eventual Consistency.
Multi Datacenter Consistency.
Optimistic Replication and Resolution.
Weak Consistency Models for Replicated Data.

Recommended Reading

[1] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based CRDTs by delta-mutation. In *Int. Conf. on Networked Systems (NETYS)*, volume 9466 of *Lecture Notes in Comp. Sc.*, pp. 62–76, Agadir, Morocco, May 2015.

[2] V. Balegas, N. Preguiça, R. Rodrigues, et al. Putting consistency back into eventual consistency. In *Euro. Conf. on Comp. Sys. (EuroSys)*, pp. 6:1–6:16, Bordeaux, France, Apr. 2015.

[3] C. Baquero and F. Moura. Using structural characteristics for autonomous operation. *Operating Systems Review*, 33(4):90–96, 1999. ISSN 0163-5980.

[4] C. Baquero, P. S. Almeida, and A. Shoker. Making operation-based CRDTs operation-based. In *Int. Conf. on Distr. Apps. and Interop. Sys. (DAIS)*, volume 8460 of *Lecture Notes in Comp. Sc.*, pp. 126–140, Berlin, Germany, June 2014.

[5] Basho, Inc. Data types, version 2.1.1. https://docs.basho.com/riak/kv/2.1.1/developing/data-types/, Viewed May 2016. https://docs.basho.com/riak/kv/2.1.1/developing/data-types/.

[6] A. Bieniusa, M. Zawirski, N. Preguiça, et al. An optimized conflict-free replicated set. Rapport de Recherche RR-8083, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, Oct. 2012.

[7] S. Burckhardt, M. Fahndrich, D. Leijen, et al. Cloud types for eventual consistency. In *Euro. Conf. on Object-Oriented Pging. (ECOOP)*, pp. 283–307, Beijing, China, June 2012.

[8] S. Burckhardt, A. Gotsman, H. Yang, et al. Replicated data types: Specification, verification, optimality. In *Symp. on Principles of Prog. Lang. (POPL)*, pp. 271–284, San Diego, CA, USA, Jan. 2014.

[9] G. DeCandia, D. Hastorun, M. Jampani, et al. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pp. 205–220, Stevenson, Washington, USA, Oct. 2007.

[10] A. Gotsman, H. Yang, C. Ferreira, et al. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *Symp. on Principles of Prog. Lang. (POPL)*, pp. 371–384, St. Petersburg, FL, USA, 2016.

[11] P. R. Johnson and R. H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, Jan. 1976.

[12] C. Li, D. Porto, A. Clement, et al. Making geo-replicated systems fast as possible, consistent when necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 265–278, Hollywood, CA, USA, Oct. 2012.

[13] N. Preguiça, J. M. Marquès, M. Shapiro, et al. A commutative replicated data type for cooperative editing. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pp. 395–403, Montréal, Canada, June 2009.

[14] H.-G. Roh, M. Jeon, J.-S. Kim, et al. Replicated Abstract Data Types: Building blocks for collaborative applications. *Journal of Parallel and Dist. Comp.*, 71(3): 354–368, Mar. 2011.

[15] M. Shapiro, N. Preguiça, C. Baquero, et al. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de Recherche 7506, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, Jan. 2011.

[16] M. Shapiro, N. Preguiça, C. Baquero, et al. Conflict-free replicated data types. In *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pp. 386–400, Grenoble, France, Oct. 2011.

[17] M. Shapiro, N. Preguiça, C. Baquero, et al. Convergent and commutative replicated data types. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (104):67–88, June 2011.

[18] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Int. Conf. on Computer-Supported Coop. Work (CSCW)*, p. 59, Seattle WA, USA, Nov. 1998.

[19] G. T. J. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Symp. on Principles of Dist. Comp. (PODC)*, pp. 233–242, Vancouver, BC, Canada, Aug. 1984.