A Study of CRDTs that do Computations

David Navalho Sérgio Duarte Nuno Preguiça
NOVA LINCS, FCT, Universidade NOVA de Lisboa

Abstract

A CRDT is a data type specially designed to allow instances to be replicated and modified without coordination, while providing an automatic mechanism to merge concurrent updates that guarantees eventual consistency. In this paper we present a brief study of computational CRDTs, a class of CRDTs whose state is the result of a computation over the executed updates. We propose three generic designs that reduce the amount of information that each replica maintains and propagates for synchronizations. For each of the designs, we discuss the properties that the function being computed needs to satisfy.

1. Introduction

Cloud infrastructures, composed of multiple data centers spread across the globe, have become central for the deployment of novel Internet services, from social networks to business applications. A large number of cloud databases have been developed in recent years, providing different level of consistency, from strong [5] to eventual consistency [2, 6, 7].

In this paper we focus on cloud databases that provide eventual consistency only. When using an eventually consistent database, applications can be made highly available by replicating the application code and data in multiple data centers and allowing a user to access any of these data centers. Low latency is achieved by routing the client requests to the closest data center and executing the request in the data center without coordinating with other data centers.

In such settings, concurrent updates may be executed in different replicas. Systems must provide a mechanism to handle concurrent updates and enforce eventual convergence of all replicas. CRDTs [10] have been proposed as a technique for helping application programmers to deal with concurrent updates. They provide eventual consistency with well defined semantics and thus make these systems more amenable to programmers. CRDTs have been adopted as a key feature in a leading cloud database, Riak, and are used in multiple large-scale systems, such as SoundCloud and Twitter's Summingbird[3].

Most CRDTs proposed in literature are replicated forms of collections. In such data types, a replica needs to maintain all data elements in all replicas. Thus, a model where every data replica maintains the same state and where all updates are propagated to all replicas is natural.

In some cases, applications are not interested in actual elements or updates, but instead on the result of a computation over them. We call computational CRDTs to the class of CRDTs whose state is the result of a computation over the executed updates. For example, a counter CRDT [10] counts the number of times an increment operation has been executed. In such cases, each replica does not need to maintain every individual update, but can instead maintain for each replica an integer that counts the number of increments executed at that replica. For synchronizing replicas, it also suffices to propagate an integer that summarizes a set of updates.

In the remaining of this paper we present a brief study of the properties of computational CRDTs. In particular, we propose three generic designs that minimize the data that needs to be maintained in each replica and that needs to be propagated for synchronizing replicas. We study the properties of functions suitable to each of the designs. Notably, our last design departs from the strict model of state-based CRDTs by the fact that the state of each replica does not need to converge, although the result of all queries executed in every replica is the same.

1.1 Related work

Aggregation techniques have been studied extensively in different settings, such as as sensor networks [11]. Our work can build on the proposed algorithms for creating replicated data types that perform computations in a cloud database.

The techniques used to model CRDTs have been used to express a distributed deterministic dataflow model for concurrent communication between processes [8]. They have also been used to provide algebraic structures for integration between batch and stream processing of aggregations [3] and to support incremental computations [9]. Unlike these works, this paper studies CRDTs that can be integrated in a cloud database as an elementary abstraction to perform computations without requiring additional support from the system.

PaPoC'15, April 21–24, 2015, Bordeaux, France.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3537-9/15/04…$15.00.
http://dx.doi.org/10.1145/2745947.2745948

The problem of optimizing information propagated for synchronizing replicas has been studied by Almeida et. al [1], who have proposed a principled approach to merge the changes produced by multiple operations and use this information to update a remote replica. In our work, the information propagated to synchronize replicas also summarizes multiple updates. However, all information is handled in the context of the CRDT. Additionally, our last design departs from the strict state-based CRDT model by allowing replicas to maintain different state.

2. System model

We adopt the CRDT state-based model [10], where replicas synchronize in a peer-to-peer way, by sending their state to other replicas, where the received state is merged with the current state. A CRDT has an interface that includes update operations that modify the state of the object. In our presentation, we define an event as an invocation of an update operation. For simplicity, we consider a single read-only operation that returns the state of the object. A CRDT includes an additional operation, *merge*, to merge a copy of a remote replica with the current replica state. In one design, we extend this model to allow a replica to send only a subset of its state to other replicas.

For fault-tolerance, we assume a crash-recovery model, where a replica that fails recovers with its state intact. In a typical cloud deployment, each data center can be seen as a single replica, although internally an object is replicated in a quorum of replicas.

3. Design 1: Incremental Computations

Our first design considers computations that can be done incrementally. In this case, computing the function over two disjoint sets of events and combining the results is equal to computing the function over the union of the two sets. Formally, a computation is incremental if there is a function fun, such that:

$$\begin{aligned}&\mathcal{F}^{\mathtt{fun}}(E_1 \cup E_2, \mathsf{hb}_{E_1 \cup E_2}) = \\ &\qquad \mathtt{fun}(\mathcal{F}^{\mathtt{fun}}(E_1, \mathsf{hb}_{E_1}), \mathcal{F}^{\mathtt{fun}}(E_2, \mathsf{hb}_{E_2}))\end{aligned}$$

where E_1 and E_2 are disjoint sets of events (operation invocations), hb_E is a partial causality order on E[1], and $\mathcal{F}^{\mathtt{fun}}$ is the function that defines the state of a CRDT that computes fun over the observed events (following loosely the formalization proposed by Burckhardt et. al.[4]).

For example, a counter with a single update operation for increment, inc, can be defined as follows:

$$\begin{aligned}\mathcal{F}^{+}_{\mathtt{ctr}}(E, \mathsf{hb}) &= |\{e \in E : e = \mathtt{inc}\}| \\ \mathcal{F}^{+}_{\mathtt{ctr}}(E_1 \cup E_2, \mathsf{hb}) &= \mathcal{F}^{+}_{\mathtt{ctr}}(E_1, \mathsf{hb}) + \mathcal{F}^{+}_{\mathtt{ctr}}(E_2, \mathsf{hb})\end{aligned}$$

For these computations, Figure 1 presents a generic CRDT design that is parameterized by the following elements: (i) V_0, the initial state associated with a replica; (ii) $\mathtt{fun}(o)$, the value of the computation for a single operation o; (iii) $\mathtt{fun}(s_1, s_2)$, the function to compose two partial results; and (iv) $\mathtt{fun}^{\mathsf{max}}(v_1, v_2)$, that returns the latest of two values.

[1] For simplicity of presentation, we drop the subscripts of hb in the rest of the paper.

In this design, each replica computes its contribution to the final value of the CRDT independently. Each replica maintains a map for the contributions of each replica. When executing an update operation, a replica updates its contribution by using function fun to combine the previous computed contribution and the contribution of the new operation (with $s[i \mapsto \mathtt{fun}(s[i], \mathtt{fun}(\mathtt{op}))]$ representing the replacement in s of the value of entry i by the new computed value). When merging two replicas, for the partial result of each replica, the most recently computed result must be kept, which is returned by $\mathtt{fun}^{\mathsf{max}}$. If the values are monotonic, it is immediate to know what is the most recent version. Otherwise, it might be necessary to maintain this information explicitly. The value of a replica is computed by applying the function fun to the contributions of all replicas.

As an example, a positive-negative counter, with an increment and a decrement operations can be defined by making:

$$\begin{aligned}V_0 &= (0,0) \\ \mathtt{fun}(\mathtt{inc}) &= (1,0) \\ \mathtt{fun}(\mathtt{dec}) &= (0,1) \\ \mathtt{fun}((p,m),(p',m')) &= (p+p', m+m') \\ \mathtt{fun}^{\mathsf{max}}((p,m),(p',m')) &= (max(p,p'), max(m,m'))\end{aligned}$$

A CRDT that computes the average of values added to an object, which could be used for example to present the average rating in a web application, can be defined by making:

$$\begin{aligned}V_0 &= (0,0) \\ \mathtt{fun}(\mathtt{add}(x)) &= (x,1) \\ \mathtt{fun}((s,c),(s',c')) &= (s+s', c+c') \\ \mathtt{fun}^{\mathsf{max}}((s,c),(s',c')) &= (s,c), \text{ iff } c > c' \\ &\quad\ (s',c'), \text{ iff } c \leq c'\end{aligned}$$

The average is computed as $^{s}/_{c}$, with (s,c) the result of the read defined in the generic CRDT design.

Other CRDTs can be defined using a similar approach, including a CRDT that computes a histogram.

4. Design 2: Incremental Idempotent Computations

In some cases, the computation to be performed besides being incremental is also idempotent. In this case, computing the function over two (potentially overlapping) sets of events and combining the results is equal to computing the function over the union of the two sets. Formally, a computation is incremental and idempotent if there is a function fun, such that for any sets of events E_1 and E_2 we have:

$$\mathcal{F}^{\mathtt{fun}}(E_1 \cup E_2, \mathsf{hb}) = \mathtt{fun}(\mathcal{F}^{\mathtt{fun}}(E_1, \mathsf{hb}), \mathcal{F}^{\mathtt{fun}}(E_2, \mathsf{hb}))$$

Replica state	Σ	$=$	$\mathbb{I} \rightarrow V$
Initial state	σ_i^0	$=$	V_0
Update op at replica i	$\texttt{op}_i(s)$	$=$	$s[i \mapsto \texttt{fun}(s[i], \texttt{fun}(\texttt{op}))]$
Read at replica i	$\texttt{op}_i(s)$	$=$	$\texttt{fun}(s[i], \forall i)$
Merge replica states	$\mathsf{deliver}\,(s, s')$	$=$	$s[i \mapsto \texttt{fun}^{\texttt{max}}(s[i], s'[i])], \forall i$

Figure 1: Generic CRDT for incremental computation.

Replica state	Σ	$=$	V
Initial state	σ_i^0	$=$	V_0
Update op at replica i	$\texttt{op}_i(s)$	$=$	$\texttt{fun}(s, \texttt{fun}(\texttt{op}))$
Read at replica i	$\texttt{op}_i(s)$	$=$	s
Merge replica states	$\mathsf{deliver}\,(s, s')$	$=$	$fun(s, s')$

Figure 2: Generic CRDT for incremental idempotent computation.

For these computations, Figure 2 presents a generic CRDT design. In this case, it is possible to keep in each replica only the computed result that is modified when executing update and merge operations.

A computation that obeys these conditions is computing the maximum of the values added to an object, which could be used in a game application for keeping the highest score. This data type could be implemented, keeping a name associated with the highest score, with names totally ordered, by making:

$$\begin{aligned} V_0 &= (-,\text{minimum value}) \\ \texttt{fun}(\texttt{add}(n, v)) &= (n, v) \\ \texttt{fun}((n, v), (n', v')) &= (n, v), \text{ iff } v > v' \lor (v = v' \land n > n') \\ &\quad (n', v'), \text{ otherwise} \end{aligned}$$

A generalization of the maximum CRDT is a top-K CRDT that keeps the K players with highest scores, which can be used to maintain a leaderboard in a game application. This CRDT can be implemented by making:

$$\begin{aligned} V_0 &= \{\} \\ \texttt{fun}(\texttt{add}(n, v)) &= \{(n, v)\} \\ \texttt{fun}(s, s') &= \texttt{max}_\texttt{k}\big(\{(n, v) \in (s \cup s') : \\ &\qquad \nexists (n, v_1) \in (s \cup s') : v_1 > v\}\big) \end{aligned}$$

with $\texttt{max}_\texttt{k}(s)$ a function that returns the k largest elements $(n, v) \in s$, with the elements ordered using the total order defined previously.

In general, this approach can be used to create CRDTs that compute a filter over the values added to the object, for which an element that does not match the filter at some moment will not match the filter at a later moment.

5. Design 3: Partially Incremental Computations

We now consider computations that are only partially incremental, in the sense that some updates observe the incremental property previously defined, while others do not. An example of such an object is a top-K object where an element can be deleted. In such cases, a value that does not belong to the top-K elements may later become part of the top, after a top element is deleted.

To address this case, a possible approach is to use a Set CRDT to maintain the set of elements that have not been deleted. In this case, all replicas maintain the complete set, and all updates need to be propagated to all replicas. The top-K can be computed locally on the value of each replica.

In Figure 3 we present an alternative approach, in which each replica maintains all operations locally executed, and each replica only propagates to other replicas the operations that might affect the computed result. Each replica maintains a set of operations and the results of the computation performed at other sites — for simplicity of notation, we assume that the result of the computation is a subset of operations. An update operation updates the local set of operations. A read operation makes the computation considering the local operations and the results of the computation at the other replicas. For synchronizing replicas, a replica sends the results of the computations to all replicas and the subset of operations known locally that can affect the computed result at other replicas (in the top-k example, a delete of an element that belongs to the top elements). When receiving the state from a remote replica, the local replica is updated by merging the local set of operations with the remote operations that may affect the result of the computation, and by registering the most recent version of the computation for each site.

A top-k replicated data type that supports an $\texttt{add}(n, v)$ and $\texttt{del}(n)$ operations can be defined as follows:

$$\begin{aligned} V_0 &= \{\} \\ \texttt{fun}(s) &= \texttt{max}_\texttt{k}\big(\{o \in s : o = \texttt{add}(n, v) \land \\ &\qquad (\nexists o' \in s : o \prec o' \land o' = \texttt{del}(n))\}\big) \end{aligned}$$

with $\texttt{max}_\texttt{k}(s)$ a function that returns the k $\texttt{add}(n, v)$ operations with largest values (n, v) for different values of n and elements ordered using the total order defined previously. $\texttt{fun}^{\texttt{max}}$ can be defined by assigning a monotonic integer to the result computed in each replica, and using this integer to decide which value is the most recent.

$$\begin{array}{lrcl}
\text{Replica state} & \Sigma & = & \big(\mathcal{P}(\texttt{op}), \mathbb{I} \rightarrow V\big) \\
\text{Initial state} & \sigma_i^0 & = & \big(\{\}, i \rightarrow V_0\big) \\
\text{Update op at replica } i & \texttt{op}_i\big((s,m)\big) & = & \big(s \cup \{\texttt{op}\}, m\big) \\
\text{Read at replica } i & \texttt{op}_i\big((s,m)\big) & = & \texttt{fun}\big(s \bigcup_{\forall j} m[j]\big) \\
\text{State to send from replica } i & \texttt{diff}\big(s,m\big) & = & (\{o \in s : \texttt{fun}\big(o \bigcup_{\forall j} m[j]\big) \neq \texttt{fun}\big(\bigcup_{\forall j} m[j]\big)\}, m[i \rightarrow \texttt{fun}(s \bigcup_{\forall j} m[j])]) \\
\text{Merge replica states} & \mathsf{deliver}\,\big((s,m),(s',m')\big) & = & \big(s \cup s', m[j \mapsto \texttt{fun}^{\texttt{max}}(m[j], m'[j])]\forall j\big)
\end{array}$$

Figure 3: Generic replicated data type for partially incremental computation.

This design enforces eventual consistency, assuming that replicas continue synchronizing until they reach an equivalent state, i.e., a state where read operations return the same result in every replica. However, this may not happen after the first synchronization step. For example, consider a top-1 object replicated in two sites: Site 1 executed operations $\{\texttt{add}(b, 15), \texttt{add}(a, 10)\}$ and site 2 executed operations $\{\texttt{add}(b, 16), \texttt{add}(c, 12)\}$. The two sites synchronize, with the top-1 element, $(b, 16)$, being known at both replicas. After this, $\texttt{del}(b)$ executes at site 1, promoting $(a, 10)$ to the top at site 1. After the propagation of $\texttt{del}(b)$ to site 2, $(c, 12)$ is promoted to the top at site 2. After the next synchronization step, the top at site 1 $(a, 10)$ is replaced by the same value as in site 2 $(c, 12)$.

6. Final remarks

In this paper we have proposed three generic designs for replicated data types that perform a computation on the operations executed by users. These designs can be used in a system that maintains CRDT replicas at multiple sites and synchronizes them using a state-based model. We present the properties that computations must obey in order to use each of the designs. These designs try to minimize the information that each replica has to maintain and propagate to other replicas for synchronization.

The last proposed design departs from the strict CRDT state-based model, while still enforcing eventual consistency. We are currently formalizing the new model and studying the relations between replicated data types implemented using this design and state-based CRDTs that implement the same functionality. In the future, we intend to study how to integrate these designs in an eventually consistent cloud database, such as Riak.

Acknowledgments

This research is supported in part by FCT scholarship SFRH / BD / 65070 / 2009, FCT projects PTDC/ EEI-SCR/ 1837/ 2012 and PEst-OE/ EEI/ UI0527/ 2014 and EU FP7 SyncFree project (609551).

References

[1] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. In *Proc. of the Third International Conference on Networked Systems (NETYS) (to appear)*, May 2015.

[2] S. Almeida, J. a. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, 2013. ACM.

[3] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proc. VLDB Endow.*, 7(13):1441–1451, Aug. 2014.

[4] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, 2014. ACM.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, et. al. Spanner: Google's globally-distributed database. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012. USENIX Association.

[6] 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 *Proc. of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007. ACM.

[7] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, 2011. ACM.

[8] C. Meiklejohn and P. Van Roy. Lasp: A Language for Distributed, Eventually Consistent Computations with CRDTs. In *Proc. of the Workshop on Principles and Practice of Consistency for Distributed Data*, Apr. 2015.

[9] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro. Incremental stream processing using computational conflict-free replicated data types. In *Proc. of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP '13, pages 31–36, 2013. ACM.

[10] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proc. of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, 2011. Springer-Verlag.

[11] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Comput. Netw.*, 52(12):2292–2330, Aug. 2008.