Constraining the Eventual in Eventual Consistency

Jim Bauwens, Florian Myter, Elisa Gonzalez Boix
Vrije Universiteit Brussel
jim.bauwens,florian.myter,egonzale@vub.be

ABSTRACT

CRDTs are highly available replicated data structures which offer strong eventual consistency in the face of concurrent operations [3]. By their definition, CRDTs eventually converge to a consistent state given enough time. However, this is not strict enough for some distributed applications. Current state-of-the-art CRDT implementations fail to provide programmers with the means to specify these constraints. As a result, programmers need to write application-level code which ignores stale or timed-out operations. In this paper, we introduce a leasing model which allows programmers to declaratively specify timing constraints for CRDTs. In short, programmers are able to attach leases to operations on a CRDT instance. When such a lease expires the underlying implementation ensures that the operation is eventually canceled for all replicas.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Software and its engineering** → **Consistency**; **Constraints**;

KEYWORDS

eventual consistency, CRDTs, leasing

ACM Reference Format:
Jim Bauwens, Florian Myter, Elisa Gonzalez Boix. 2018. Constraining the Eventual in Eventual Consistency. In *PaPoC'18: 5th Workshop on Principles and Practice of Consistency for Distributed Data , April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3194261.3194263

1 INTRODUCTION

CRDTs are data structures which are replicated across a distributed application, and can be concurrently updated without requiring consensus or distributed transactions. To this end, CRDTs constrain the type of operations which can be applied to them to be commutative, associative and idempotent. By definition [3] CRDTs guarantee strong eventual consistency. Assuming no new updates happen to a set of replicas, they will eventually converge to the same state without conflicts.

PaPoC'18, April 23–26, 2018, Porto, Portugal
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5655-8/18/04…$15.00
https://doi.org/10.1145/3194261.3194263

In theory, it may take an infinite amount of time for replicas to converge. But this may not be suitable for many distributed applications which require operations to be executed within certain deadlines. Applications might choose to disregard certain operations on a CRDT if they fail to converge at all replicas within a certain time frame. However, using state-of-the-art CRDT implementations, programmers are unable to specify such application-specific time constraints.

Manually keeping track of timing constraints on CRDTs is a tedious task, as they are a cross cutting concern over the entire application. The programmer needs to keep track of all operations on each replica in order to cancel operations which exceed their time constraint. Moreover, canceling operations requires application-specific protocols.

In this paper, we propose to incorporate a leasing model into the CRDT abstraction, resulting in *leased CRDTs*. The key idea is to enable programmers to declaratively specify timing constraints on operations as leases. If all replicas fail to observe a leased operation within the specified time frame we say that the operation fails. Subsequently the operation is revoked for all replicas.

2 MOTIVATING EXAMPLE

In this section, we motivate the need for leased CRDTs by means of an illustrative example application. Consider a mobile auction application where a user can browse a list of items and add items deemed interesting to a wish list. The wish list is implemented as a CRDT and is synchronized with an online platform. Items are sold at specific times to the highest bidder and are removed from the item list afterwards.

Since a mobile device may not have permanent access to the Internet, operations pending on the wish list CRDT might not yet be synchronized with the online platform. This means that some auctioned items related to the operations might already be sold before all replicas have converged. In this case the operations are no longer interesting and reverting them is an acceptable step.

By adding a lease on the operations (based on the auction time) this reverting can be done automatically. Reverting the addition of an item to the list is done by issuing a remove operation of the specified item. We say thus that `remove` is the *anti-operation* of `add`.

3 LEASING AND CRDTS

Before describing leased CRDTs, we introduce the necessary background information and terminology on leasing and CRDTs.

Leasing has been used in distributed systems to limit access on resources for a specific duration of time [2]. In particular, a resource owner (also know as the *lease owner*) grants a resource user (the *lease grantor*) usage of the resource under a contract that specifies the lease term and conditions under which the lease is valid.

CRDTs offer a set operations that can be applied where it is mathematically proven that concurrent updates on different replicas of the CRDT will not result in conflicts.

Typically, these operations can be either used to query the CRDT or to update it. Update operations are performed in two phases. First, an *at source* part is executed only on the replica that is the source of the operation. The purpose of this phase is to prepare the actual updating of the CRDT. Second, a *downstream* part is executed on all replicas, based on information computed by the *at source* part [3]. For example, the *at source* part of the add operation of an OR-Set CRDT computes a unique identifier, which is then used in the actual modification on all replicas during the *downstream* phase. Note that some CRDTs do not need an *at source* part, e.g. a simple counter CRDT.

4 A LEASING MODEL FOR CRDTS

In this paper, we propose a leasing model for CRDTs that allows developers to apply timing constraints to the set of operations offered by a CRDT. In our approach, leasing is applied to the *downstream* phase, given that this is where the actual replication happens. More concretely, for every replica there is one lease per downstream phase. We say that a lease *expires* if its downstream phase is unable to complete before the specified time deadline. This means a lease expires if at least *one* replica does not observe the downstream phase on time.

Since a lease may expire while some replicas already observed the downstream phase, countermeasures need to be put in place so that all replicas remain consistent. In our work, we propose to associate an *anti-operation* for each operation defined on a CRDT.

An anti-operation is the inverse of an associated operation, i.e. applying it equates to rolling back the original operation. For example, in the case of a counter CRDT, the anti-operation of an increment is a decrement.

Anti-operations are often part of the set of operations on CRDTs. For example, counters and sets have for most versions pairs of operations that match our definition of anti-operations *to a certain degree*. In particular, the add/remove operations of a set and the increment/decrement operations of a counter can be considered anti-operations from one another.

Note, however, that a pairing operation is in most cases not exactly the correct anti-operation. For example, with an OR-Set it would be incorrect to use `remove(x)` after an `add(x)` as anti-operation. This is because the `add` operation actually adds a tuple to the dataset, containing `x` and a uniquely generated ID. Removing `x` would remove all tuples in the dataset that have `x` as first item. The correct anti-operation for `add(x)` would be to remove only the tuple that has `x` along with the correct ID.

In conclusion, leasing enables developers to specify deadlines with regards to the time it takes for CRDTs to become strongly eventually consistent. However, in order to ensure this, it is crucial that developers specify correct anti-operations.

4.1 Leased CRDTs in Action

This section describes two small applications that use counter CRDTs with timing constraints on the increment operation. The first version uses a leased CRDT while the second manually manages the deadlines. Both versions are written in LuAT[1], a Lua library we implemented for distributed programming which incorporates the concepts of Ambient-oriented Programming [1].

Listing 1 shows how to use the leased CRDT framework built in LuAT. `LeasedCounterCRDT` takes a string representing a nominal type used for other nodes in the network to discover this CRDT, and a callback function applied when the CRDT is updated. It then creates a counter CRDT instance which can be discovered in the network by means of the `shared_counter` string. Line 4 shows how operations are leased in a CRDT: they are basically augmented with a parameter to specify the lease time. If the `increment` operation fails to be replicated within the specified time, the CRDT framework automatically perfoms a rollback.

```lua
1  local counter = LeasedCounterCRDT("shared_counter", function (value)
2        print('Counter updated', value)
3  end)
4  counter:increment(10, seconds(3))
```

Listing 1: Using a leased counter CRDT

Listing 2 uses a manual approach to add timing constraints to a simple counter CRDT as the one offered by traditional CRDT frameworks. Such a CRDT is created on line 12 by means of the `CounterCRDT` construct, which takes as arguments a string and callback function such as `LeasedCounterCRDT`. However, without leased CRDTs, programmers need to take care of two kinds of bookkeeping code. First, code for registering deadlines and periodically checking if they have elapsed needs to be written (lines 1 to 11). Second, programmers also need to manually link the CRDT `increment` operation with a deadline and encode compensating actions if the deadline has passed before convergence (lines 19 - 28). Moreover, the timing constraints are only checked at the source of the operation while in the leased CRDT version the deadlines are validated consistently over all replicas by the framework. Note also that this boilerplate code may need to be written over and over for each type of CRDT required in an application as the code responsible for rolling back is application-specific.

```lua
1  local TIMEOUT = 3
2  local deadlines = {}
3  framework.addInterval(100, function ()
4      local time = system.getTime()
5      for future, deadline in pairs(deadlines) do
6          if time > deadline then
7              future:ruin()
8              deadlines[future] = nil
9          end
10     end
11 end)
12 local counter = CounterCRDT("shared_counter", function (value)
13       print('Counter updated', value)
14 end)
15 function increment_counter(n)
16   local deadline = system.getTime() + TIMEOUT
17   local future = counter:increment(n)
18   future
19   :whenBecomes(function()
20     local time = system.getTime()
21     if time > deadline then
22         counter:decrement(n)
23     end
24   end)
25   :whenRuined(function()
26       counter:decrement(n)
27   end)
28   deadlines[future] = deadline
29 end
30 increment_counter(10)
```

Listing 2: Using a manual approach for time constraint management

[1]https://git.infogroep.be/jibauwen/LuAT/

5 CONCLUSION

CRDTs are data structures which allow for concurrent operations on replicated data in distributed systems. Moreover, they guarantee that eventually all replicas end up in the same state. However, this convergence might take an infinite amount of time. We showed by example that some applications may require this convergence time to be constrained. However, implementing these constraints using state-of-the-art CRDT implementations is cumbersome. In this paper, we propose *leased CRDTs* as a novel programming construct which enables programmers to declaratively specify convergence timing constraints.

REFERENCES

[1] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. 2007. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. In *Chilean Society of Computer Science, 2007. SCCC '07. XXVI International Conference of the.* 3–12. https://doi.org/10.1109/SCCC.2007.12

[2] C. Gray and D. Cheriton. 1989. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. *SIGOPS Oper. Syst. Rev.* 23, 5 (Nov. 1989), 202–210. https://doi.org/10.1145/74851.74870

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Technical Report 7506.