Sequential writes as a convergent replicated datatype

Christian Weilbach, Konrad Kühne and Annette Bieniusa

July 4, 2015

Abstract

1 Motivation

The following work introduces a new convergent replicated datatype (CRDT), we call "*geschichte*" (meaning history in German), which retains the order of write operations at the tradeoff of after-the-fact conflict resolution. It evolved as part of a replication system to share data openly on the web. TODO cite replikativ. One goal of this replication system is to model a *git-like* interaction workflow with application state. This allows to program web applications with distributed state in a similar fashion to native applications and is well understood by many programmers. cite HISTODB

Most importantly it regains sequentially ordered writes in decentralized distribution system under the tradeoff of after-the-fact conflict resolution. This allows the replication mechanism to distribute all values without conflicts, but conserves them in the value of the newly introduced datatype. That way conflicts in replication turn into conflicts in the value of data and can be reasoned independently from network topology and propagation of state changes. The newly introduced sequential datatype makes application development conceptually easier as we show in the evaluation of our prototype of a social network application.

2 Related Work

A lot of prior work in the field of distributed systems exists. We will have a look at it from the two angles which motivated the design of the new datatype: distributed revision control systems (DVCSs) and convergent replicated data types (CRDTs).

2.1 DVCS

Most code today is versioned with *well-designed* and *mature* distributed revision control systems like `git`, `mercurial` or `darcs` and has inspired light-weight open-source friendly workflows, e.g. through github TODO quote. Operations on the repository in these systems can be executed offline and are synchronized explicitly by the developer/user. In terms of the CAP theorem TODO these systems are always available but not consistent. They provide some means for after-the-fact conflict resolution of text files, typically through some diffing TODO mechanism. While this has proven very effective, attempts to transfer it to data have had limited success so far. Examples are filesystems built on top of git like gitfs or git-annex. There have also been repeated attempts at using git to implement a database. We think that these attempts while close to our work and interesting are doomed to failure, because they try to generalize a manual low frequency write workload of development on source code files with differently evolving write-workloads of state-transitions in databases.

Some of the problems in these systems on top of DCVS are:

- can be used for data, e.g. `JSON`, but these force *line-based text-files* in a filesystem structure to be compatible with the default diffing mechanism for conflict resolution
- file systems are the historic data storage modelling a non-distributed low-level binary view on data within a single hierarchy (folders), and hence cannot capture and exploit higher-level structure of data to resolve

conflicts and replicate efficiently controlled by the application.

- often scale *badly* with *binary blobs* as they take part in the diffing step. They then need an out-of-band synchronization like git-annex.

Instead we reimplement the important concepts of DCVS on top of explicit, well-defined distributed datatypes to keep guarantees in convergence and scalability in our replication system. Most importantly we can use other more efficient commutative datatypes for write-intensive parts of the global state-space, e.g. posts in the social network and indizes on hashtags. To cover this we have implemented a general synchronization system for CRDTs which is augmented with our datatype for scenarios with strong consistency needs. This means applications need to be build with an explicit concept of datatypes for distributed write-operations in mind as this cannot be bolted on top of e.g. a filesystem like AFS, glusterfs, Coda or gitfs after the fact without considerations from the application write workload. All these filesystems for instance do not reasonably allow to run ACID sql-databases on top, because their consistency demands cannot be satisfied.

2.2 CRDT et al. research for similar datatypes (references)

- techreport Shapiro et al. 2011
- cloud datatypes Burckhardt et al. 2012
- mergeable datatypes Lorenz and Rosenan 2014

3 Model

3.1 git-like repository

3.1.1 define commit graph representation

- commit graph: causal-order and branches data-structure

```
{:causal-order {10 [], ;; root
                20 [10],
                30 [20],
                40 [10],
                50 [40 20]}, ;; two parents to me
 :branches {"master" #{30 40},
            "merged" #{50}}}
```

- `causal-order` is a *growing* graph without removals
- `branches` point to *tips* in this graph
- *branch heads* are a set
- visualize?

3.2 operations

- before/after plots

3.2.1 commit

- commits a new value (transaction + argument)

3.2.2 branch

- create a new branch given a parent
- no visualization needed

3.2.3 pull

- pull all missing parent commits from remote-tip into branch.
- visualize added subgraph (missing parents)

3.2.4 merge

3.2.5 TODO graph plots

- which ones? how?
- before-after

3.3 CRDT specifications

- techreport p.6

3.3.1 TODO upstream

- same operations as above in terms of crdt: upstream

3.3.2 downstream

- only "downstream" op is *merging* ops/state
- *remove stale parents* through `lowest-common-ancestor` (lub) search
- *multiple branch heads* can *safely* occur at *any point* of propagation
- conflict is part of the value, not of datatype

3.3.3 TODO crdt proof

- guaranteed state synchronisation on connection (costly, but is automatically optimized by efficient state serialization on reconnect)
- conflict free: upstream, downstream
  - graph no problem, grow-set, can have no problems because hashes are like inline values in hash-map
  - need to show that heads always correct; upstream correctly adds heads in each case; downstream uses lca to clean them up on every op
- too many heads => expensive, lca, solutions?
- lca description

4 Evaluation

4.1 Example application

- profile management topiq

4.2 TODO think about it

- combination with other "value"-conflict-free crdts
- x-crdt

5 Conclusion

References

[Bur+12] Sebastian Burckhardt et al. “Cloud Types for Eventual Consistency”. In: (2012).

[FB99] Armando Fox and Eric A. Brewer. “Harvest, Yield and Scalable Tolerant Systems”, Proc. 7th Workshop Hot Topics in Operating Systems”. In: (1999).

[LR14] David Lorenz and Boaz Rosenan. “Versionable, Branchable, and Mergeable Application State”. In: (2014).

[Sha+11] Marc Shapiro et al. “A comprehensive study of Convergent and Commutative Replicated Data Types”. In: (2011).