A convergent replicated datatype for a git-like repository

Christian Weilbach, Konrad Kühne and Annette Bieniusa

July 8, 2015

Abstract

1 Introduction

The following work introduces a new convergent replicated datatype (CRDT), which we call `geschichte` (German for *history*). It 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 develop applications with distributed state in a similar fashion to native applications with exclusive local state.

Most importantly `geschichte` regains sequential ordering in a decentralized system with distributed writes. This allows the replication mechanism to distribute all values without *replication level* conflicts, but conserves them in the value of the newly introduced datatype. That way conflicts in replication turn into conflicts in the value of the datatype and can be reasoned about locally, i.e. 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.

1.1 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). Similar to these systems the datatype is always *available*, but *eventual consistent*. For a general overview of consistency conditions, have a look at Dziuma, Fatouru, and Kanellou 2013.

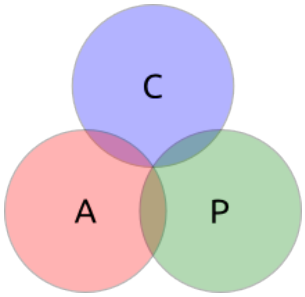

Figure 1: While one cannot have consistency, partition-resistance and availability, many tradeoffs are possible. n.d.

1.2 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 software development workflows. Operations on each repository in these systems can be executed offline and are synchronized explicitly by the developer/user. In terms of the CAP conjecture Fox and Brewer 1999 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 for source code, 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` TODO. There have also been repeated attempts at using `git` directly to implement a database. We think that these attempts, while being close to our work and interesting, are doomed to failure, because they try to generalize a highly optimized workflow of a manual low frequency write-workload for development on source code files to differently evolving write-workloads of state-transitions in databases. Much better tradeoffs can be achieved by picking the important properties of a DVCS and composing them transparently within the application. This then allows to build scalable, write-workload oriented datatypes on the application level.

Some of the problems in systems built on top of off-the-shelf DVCS are:

- they can be used for data, e.g. in `JSON` format, but this requires serialization in *line-based text-files* in a filesystem structure to be compatible with the default diffing mechanism for automatic conflict resolution. When the diffing of text files is customized in any of these DVCSs, almost a complete reimplementation of operations, e.g. for `git`, becomes necessary and the desired compatibility is lost. It is in our view much more reasonable to take out the concept of text files completely and introduce an algebra of simpler datatypes. TODO cite logoot

- file systems are the *historic* data storage model for 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 model and resolve conflicts. Today

the preferred way to model state from an application developer perspective is often a relational model or language-specific datastructures as they are very declarative and allow to focus on data instead of filesystem implementation details.

- often scale *badly* with *binary blobs* as they take part in the underlying diffing step. `git` then needs an out-of-band synchronization like `git-annex` to compensate, further complecting replication.

Instead we reimplement the important concepts of a DVCS on top of *explicit*, *well-defined* replicated 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 indexes on hashtags. A DVCS introduces considerable overhead on these operations. To cover this we have implemented a general synchronization system for CRDTs which is augmented with our datatype for scenarios with strong consistency needs.

Our replication system also allows atomic updates on compositions of datatypes, allowing strong consistency for important parts of the state while unblocking update propagation on *commutative*, *write-intensive* parts of the application state. 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 of the application specific *write workload*. All these filesystems for instance do not reasonably allow to run ACID databases on top, because the consistency demands cannot be satisfied and conflicts showing up on the filesystem level cannot be properly resolved with binary blobs.

1.3 Convergent replicated datatypes (CRDTs)

While the original motivation for the datatype was to implement a DVCS-like repository system for a ACID database in an open and partioned environment of *online* and *offline* webclients and servers, a synchronisation mechanism was lacking. While this might seem possible to implement later, DVCS systems like `git` don't allow propagation of conflicts (multiple branch heads) and hence have no proper replication protocol. These conflicts can show up in any part of the network topology of replicas during propagation of updates and they can only be resolved supervisedly. Since the system has to stay available and replicating to scale and be failure resistant, we decided to build on prior work on convergent replicated datatypes Shapiro et al. 2011. CRDTs fulfill our requirements as they don't allow and need any central coordination for synchronization. They also provide a formalism (algebra) to specifiy the operations on the datatype and prove that the state of each replica always progresses towards global convergence. CRDTs have found application e.g. in `Riak` to allow merging of the network state after arbitrary partitions without loss of write operations. This is achieved by application of so called *downstream* operations on the state

of the CRDT. These operations propagate as messages through the network monotonely increasing in time (messages arrive in order between replicas). While this fits the replication concept, it does not provide strong consistency for sequential operations.

The notion of a CRDT in general implies automatic mergeability of different replicas and does not allow conflicts which then would need some centralized information to be resolved. Hence they are also referred to as *conflict-free* replicated datatypes. `geschichte` breaks with this notion by merging conflicts (branch heads) into the value of the datatype. This allows resolution of the conflict at any point in the future on any replica. CRDTs so far have mostly captured commutative operations on *sets*, *counters*, *last-write wins registers*, growing *graphs* and domain-specific datatypes e.g. for *text editing* Shapiro et al. 2011. Necessarily none of these prior datatypes allows to consistently order distributed writes. These CRDTs nonetheless have benefits compared to our repository datatype, because they cause less overhead on synchronisation and don't require conflict-resolution on application level, provided commutativity of the datatype operations is acceptable. We hence generalized our replication with a CRDT interface and reformulated our datatype in terms of this interface.

Similar concepts of datatypes to CRDTs exist, there has been for instance the development of *cloud datatypes* Burckhardt et al. 2012 which similar to CRDTs try to raise the datatype interaction level of commutative write operations to the application. The design still happens from a cloud operator's perspective though, as their *flush* operation allows *explicit* synchronisation with some *central* view on the data on a cloud server. All their non-synchronized datatypes can be implemented with commutative CRDTs. TODO read newer papers

Close to our work are versionable, branchable and mergeable datatypes Lorenz and Rosenan 2014. This work models the datatypes with an *object-oriented* approach as a composition of *CRDT-like* commutative datatype primitives (e.g. sets). To resolve conflicts each application has to compose the state with a custom datatype which knows how to resolve conflicts in an *application level* way. They demonstrate this with a hotel-booking system, which avoids overbooking. Similar to traditional CRDTs their datatypes require automatic conflict resolution during the replication process. Furthermore since each state is modelled as an application specific datatype, the code for conflict resolution has to be provided consitently to each peer participating in replication. Having general datatypes and compositions thereof in contrast allows us to replicate without knowledge of the application and to upgrade the replication software of the CRDTs more gradually, independent of application release cycles. It also means that all peers can participate in the replication no matter if they are assigned to an application or not.

2 Model

2.1 git-like repository

2.1.1 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 merge
 :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?

2.2 operations

- before/after plots

2.2.1 commit

- commits a new value (transaction + argument)

2.2.2 branch

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

2.2.3 pull

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

2.2.4 merge

2.2.5 TODO graph plots

- which ones? how?
- before-after

2.3 CRDT specifications

Data: this text
Result: how to write algorithm with LaTeX2e
initialization;
while *not at end of this document* **do**
    read current;
    if *understand* **then**
        go to next section;
        current section becomes this one;
    else
        go back to the beginning of current section;
    end
end

Algorithm 1: How to write algorithms

- techreport p.6

2.3.1 TODO upstream

- same operations as above in terms of crdt: upstream

2.3.2 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

2.3.3 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

3 Consistency scenarios

TODO more explicit scenarios? (with graphics)

Since the major difference of `geschichte` compared to commutative datatypes is the decoupled *value-level* conflict resolution, we now want to explore how this can be used to gain different degrees of consistency in applications.

3.1 Strong consistency

As a benchmark for *strong* consistency we consider the transaction log of a typical *ACID* relational database. Such a transaction log cannot be modelled by automatically merging datatypes in a system with distributed writes, since merges of *non-commutative* operations potentially alter the history of transactions. No consistency guarantees on the current state of the database can be given then, since any of the non-commutative operations could still be affected by some unsynchronized peer.

In any system, e.g. a trivial one consisting only of growing sortable sets, strong consistency can be modelled by having a single writer with a singular notion of time serializing the access to the transaction log (set) and rejecting transactions which would conflict. This is also the explicit design decision in `Datomic`, one inspiration for this work. We can cover this scenario by allowing commit or *non-conflicting* pull operations on a *single* peer. Note that it might be internally distributed on a strongly consistent shared memory, e.g. in different data-centers. Modelling this with a branch in the repository is straightforward as it then can never be in a conflicting state.

The interesting new choices are possible when different peers might commit to a branch and the decoupled conflict resolution comes into play. In these cases conflicts can occur, but they might still be resolvable due to application level constraints or outside knowledge.

3.2 Data moderated consistency

Similar to the hotel booking scenario in Lorenz and Rosenan 2014 we can allow to book a room optimistically and then have *one* repository in the system updated strongly consistently on a peer which selectively pulls and merges in all changes where no overbooking occurs. It then provides a globally consistent state and actively moves the datatype towards convergence. The advantage of the `geschichte` datatype is that this decision can be done locally on one peer, independent of the replication. Importantly, since the decision happens again in a controlled, strongly consistent environment, it can happen supervisedly and arbitrarily complex decision functions can be executed atomically. This logic and control would be unpractical to distribute on each peer as is done in Lorenz and Rosenan 2014. Assume for example that the preferences of a user in a different CRDT or database allow rebooking rooms in a comparable hotel nearby. In this scenario the pulling operation can decide to apply further transactions on the database to book rooms in another hotel depending on information distributed elsewhere instead of just rejecting the transaction. Furthermore part of this information could be privileged and outside of the replication

system, making it impossible in a system of open replication like ours to automatically merge values on every peer.

3.3 User moderated concistency

In our current replication system each user can commit to the same `geschichte` repository on different peers at the same time (only affecting his own consistency). In this case the user takes the position of the central agency providing consistency. We can take a private addressbook application as an example. In this case we can optimistically commit new entries on all peers, but in the case where the user edits the same entry on an offline and later on an online replica, a conflict will pop up once the offline replica goes back online. Automatic resolution is unreasonable, because the integrity of the entry can be provided by the user without data loss. Since these events are rare, user-driven conflict resolution is the best choice and can be implemented by the application appropriately in a completely decentralized fashion.

3.4 Optimistic consistency

In general operations which need consistency but have low frequency, e.g. operations on bank accounts, can be written optimistically under the assumption that few conflicts arise. They still need to be confirmed by some central peer, for instance by a simple non-conflicting pull-hook, but the expensive logic and consistency conditions can be determined on each peer locally, while the central peer only determines order and rejects merges. Whenever an operation needs confirmation, the repository of the centrally pulling peer can be tracked whether the operation succeeded. For such scenarios the long-term average frequency of write-operations needs to be coordinated with the central peer. Besides this latency information, no network topology needs to be known.

4 Evaluation

4.1 Example application

- profile management topiq?
- addressbook
- todo-app?
- accounting?

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).

[DFK13] Dmytro Dziuma, Panagiota Fatouru, and Eleni Kanellou. “Survey on consistency conditions”. In: (2013).

[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).