Practicability of CRDTs in Application Development

Raymond Xu and Nigel Schuster

Abstract— **Conflict-free Replicated Data Types are a special data structure that can be merged deterministically and can therefore work in a distributed system without a need for complex coordination protocols. The concept is very appealing due to its mathematical elegance, high availability offering, and strong eventual consistency semantic, but has not been popular in industry to date as a database solution. Due to this striking discrepancy between literature and practice, the benefits and disadvantages of using CRDTs in modern application development is an interesting area to explore. By migrating components of an open-source application from a traditional relational database to a CRDT-based one, we perform a hands-on analysis of CRDTs as a data paradigm. In this paper, we examine the trade-offs of using CRDTs in general application data stores, point out specific problems that we observed, and highlight specific use cases that CRDTs are best suited for.**

I. Introduction

CRDTs, or Conflict-free Replicated Data Types, are special data structures that, when manipulated by a specific set of legal operations, can be deterministically merged, always resulting in convergence. This allows an eventual consistency semantic to be obtained without complex coordination protocols between replicas, and also supports high availability. However, due to the required properties of CRDT operations such as idempotency and commutativity, there only exist a handful of public, large-scale systems built on top of CRDTs. The largest industry examples include the League of Legends friends list system which is built on top of Riak, a database with CRDT data types, and SoundClouds Roshi, a LWW CRDT set built on top of Redis, which SoundCloud uses to back their news feed. Because of the particular semantics that CRDTs impose on the consistency of the data and the total set of legal operations on the data, it is clear that some application data models or use cases are incompatible or difficult represent exclusively with CRDTs.

Our project is to explore the extent to which CRDTs can be applied to modern application development by migrating an existing open-source application from a traditional SQL data store to one backed by CRDTs. When we expectedly run into complications or inconsistencies between the semantics available in CRDTs and the desired application semantics, we will explore solutions or workarounds to the problem and document our learnings. After migrating various feature sets to CRDTs we will not only analyze the feasibility and limitations of our implementation details and hope to uncover non-trivial differences in the semantics of specific use cases in doing so. We will compare our findings to current implementations of CRDTs in the industry, and conclude the discussion by discussing specific use cases that are best suited for CRDTs.

II. Existing Literature

A. Fundamentals

The majority of existing literature is focuses on theoretics, such as different types of CRDTs and different implementation semantics. CRDTs were largely made useful in 2011 in Shapiro et al's "A comprehensive study of Convergent and Commutative Replicated Data Types" [1]. This landmark paper formalizes the notion of achieving eventual consistency using asynchronous object replication and outlines the requirements for CRDTs. The key requirement is that CRDTs must contain a merge function that is associative, commutative, and idempotent. Given these properties, it can be shown that convergence follows. The paper presents a large number of CRDTs designs such as registers, counters, sets, and graphs, and present the two techniques of state-based replication and operation-based replication. They also tackle the problem of garbage collection, as CRDT implementations may produce unbounded growth if not done carefully. Below is a select portion of the

suite of basic CRDT designs that they outline in their paper.

- G-Counter: Grow-only counter
- PN-Counter: Supports decrement as well
- Non-negative Counter: Supports decrement but is always non-negative
- LWW-Register: Last write wins determined via timestamp
- G-Set: Grow-only set
- 2P-Set: An element can be removed but then it can never be added back
- LWW-Set: Last write wins determined via timestamp
- OR-Set: Observed-Remove Set where the outcome of a sequence of operations depends on causal history
- Add-only monotonic DAG: a DAG where an edge may be added only if it points in the same direction as an existing path

Shapiro et al. also suggest that an application could use an OR-Set to implement a shopping cart, which maps a unique id number to the quantity in the cart [1]. Adding an item to the shopping cart corresponds to an add operation, and canceling or checking out corresponds to the remove operation.

B. Operational Transform

A good amount of literature is dedicated to the topic of operational transform, which powers real-time and offline collaborative tools such as Google Docs. This is out of scope for our project as we will focus on CRDTs as a data store for general application development.

C. A New JSON CRDT

More recently in 2017, Martin Kleppmann and Alastair R. Beresford present a design for a generalized JSON datatype built to conform to CRDT properties [3]. Because JSON is an extremely common data format in modern application development, being the de-facto format for the majority of APIs, data stores, and communication layers, Kleppmann and Beresford hoped that developing a formalized JSON CRDT type would expand the practical implementation possibilities for CRDTs in general. Their contributions tackle the nesting problem. Since JSON must support arbitrary nesting of ordered lists and maps to be useful and practical, which previously had not been solved by existing algorithms, a novel algorithm had to be created. They tediously outline a formal merge operation that utilizes Lamport timestamps, a series of tree traversal rules, and specific state handling to achieve convergence. However, they acknowledge that while their semantic does indeed preserve convergence towards the same state, it may produce outcomes that are surprising to the programmer at a first glance. Additionally, they are currently working on implementing their formal mathematical model and intend on publishing an analysis of it afterwards. At the time of writing, their implementation has not been published yet.

III. Design

A. Overview

We have elected to not pursue building an application from scratch using CRDTs, but instead migrate an existing application to CRDTs. This has the advantage of letting is compare two solutions without investing extra effort. Additionally, in industry it is far more common to migrate an existing code base to new technologies than it is to start new projects from scratch. By attempting to perform an application data migration ourselves, which is inherently challenging, we will be forced to work directly in the data layer and understand more of the nuances of CRDTs.

B. CRDT Store

Our CRDT store of choice is Riak KV. Riak is a product line of distributed databases from Basho Technologies and is comprised of Riak KV, Riak TS, and Riak S2. Riak KV is most suitable for our use case of managing general application data. Riak KV is a distributed NoSQL database with in-built support for CRDT data types [2]. Additionally, they support cluster replication and high scalability using commodity hardware. We selected Riak because it is the most high-profile CRDT store in industry today, being used by companies at large scales such as GitHub, Comcast, Riot Games, and more. This makes our project implementation and our research findings grounded in the practicality, as if you were to build or migrate and application to CRDTs you would use Riak or something similar. Riak provides a

client library in Python that we will leverage in our development process.

C. Application

The application we have chosen to migrate is called Vataxia[1][2]. Since we are performing a data migration, we have frozen the frontend code, and will attempt to exclusively modify the backend and replicate the same client experience. The backend is written in Django and leverages the Django REST framework. The default database Vataxia uses is PostgreSQL, a traditional open-source relational database that is very popular in industry right now. We will cut out the Postgres layer and replace it with code that talks to Riak.

D. Containerization

The application structure we are working with is non-trivial and fairly complex as it resembles a small micro-service architecture. The first component is the Vataxia backend that uses Django and various Python packages. The second component is the Vataxia frontend that uses React and Webpack. The third component is a Riak cluster that needs orchestration, the custom addition of CRDT data types, and cluster configuration such that our Riak client in the backend can talk to it. After manually performing the setup on our local machines and verifying our development environment, we have taken the effort to containerize our entire application project using Docker and Docker Compose. Now anyone can spin up all the services in our project using simple Docker commands, independent of their operating system or local environment settings. We spent a few weeks working on accomplishing this containerization because we would like our project to be easily reproducible by others and we intend for our research to be applicable to real-world use cases, and containerization is a common industry standard nowadays.

IV. Implementing Voting

Our first challenge was to implement voting on posts with CRDTs. We saw this as a good target, since voting is a very simple process.

[1]https://github.com/buckyroberts/Vataxia

[2]https://github.com/buckyroberts/Vataxia-Frontend

A. Existing Implementation

In the existing implementation a vote was represented by a table containing the post id, the user id and negative or positive 1. This allowed us to use simple SQL aggregation to calculate the vote tally per post. A user also had the option to rescind his vote. The implementation simply deleted the table entry when a vote was rescinded. If a vote was changed from up to down or conversely, then the existing table entry was modified.

B. Implementation Approaches

CRDTs offered us a multitude of options to rewrite the voting. We will first present different approaches for the implementation and then discuss strengths and weaknesses.

1) Simple Counter: The simplest possible implementation we considered is to maintain a counter on every post for every upvote and a separate counter for every downvote. In the end effectively we simply subtrace the the negative counter from the positive counter to have the vote count.

2) PN-Counter: Some CRDT implementations, including Riak implement a more advanced Positive-Negative Counter (PN-Counter). Each post is assigned one PN-Counter to keep track of votes. The approach is effectively the same as IV-B.1, but relied on more proven datatypes and required less work for us.

3) G Set: We considered using two Grow-only Sets (G-Set) on each post, effectively mimicking IV-B.1, but maintaining a list of people who up or downvoted a post.

4) 2P Set: Building on IV-B.3 we considered using the Two-Phase Set (2P-Set) instead of a G-Set. This set internally maintains a second tombstone set, therefore a user is able to rescind his/her vote.

5) LWW-Element Set: An alternative to the 2P-Set is the Last-Write-Wins-Element-Set (LWW-Element-Set). The LWW-Element Set allows to repeatedly change the vote, effectively a user can upvote, rescind the vote and upvote again, if he/she changed his/her mind. To achieve this, a LWW-Element Set timestamps every operations, effectively versioning data.

6) User Set: We also considered storing the voting data on the user entity instead of the post. This would reduce issues if a user chose to delete his account, since all data associated with him must also be removed.

C. Problems of each Approach

All approaches have validity to them, but come with different tradeoffs. Each approach we presented in section IV-B has a problem. In the following we will point out each problem, so that the reader can understand the challenges of the different approaches.

1) Counters: The approach of using counters as presented in IV-B.1 and IV-B.2 is fundamentally flawed. A counter alone does not provide the notion of who voted. Effectively we lose relevant information. A user will be able to repeatedly vote, since we are not tracking who voted.

2) Sets: We presented various approaches using sets. First, a grow only set as presented in IV-B.3 shows two major problems:

(a) Since we are using a grow only set, once a user has voted, he is not able to rescind his vote. From a usability perspective this might be acceptable, but we aim to match the implementation presented before.

(b) A user can upvote and downvote at the same time. Since we maintain two sets, upvotes and downvotes are maintained in separate sets, we do not have any consistency constraints that the two sets may not intersect. The constraints may be enforced on the application level, however it would possibly involve using transactions. Our core belief is that CRDTs are able to reduce overhead by eliminating the need for coordination. Therefore we try to avoid transaction whenever possible.

Next, we consider the 2P-Set from IV-B.4. It partially addresses (a) by allowing users to remove their vote after it was casted. However we notice, that it will not allow users to vote again after rescinding their vote. Further, a 2P-Set also does not address the problems (b) in any way. Our final approach IV-B.5, a LWW-Element-Set, solves (a) fully, since it allows to freely cast and remove votes. However, we have to note how this is achieved. Our database, Riak, uses Vectorclocks to determine a timestamp. We note, that this solution is the best that can be achieved with a set, but it bears to major flaws:

- A Vectorclock requires coordination. While this issue is not as significant as transactionality, we want to avoid as much interprocess communication as possible.
- A LWW-Element-Set ”cheats” the system by relying on a vector clock for coordination, yet a fundamental idea of CRDTs is that a local operation can be propagated at any point in time to other machines. Effectively it relies on being able to communicate with other machines to determine a timestamp. It is not able to work in a split-brain scenario.

Finally, we also note, that it does not address (b): The consistency challenges persist.

We appreciate the idea of maintaining a set of posts with votes, as described in IV-B.6, but discarded it due to practicality concerns. Aggregating the votes per post would require a scan over all users.

D. Maps

A less discussed datatype for CRDTs is maps. Any set can also be turned into a map, so that each key of a set also has some data attached to it. Each field itself can have different merge strategies depending on the datatype of the field. We decide to treat each vote as a boolean, where true is an upvote, false is a downvote and the absence of the field indicates no vote is casted. Riak supports two flags and registers. Former will let votes win over disable, latter is based on vector clocks and lets the last write win. The possible deletion of a field is resolved on the level of the map datastructure: Concurrently, add/update wins over removal. Without concurrency, a deletion will succeed.

Maps address the critical requirement of consistency from (b). By having a single datapoint per post and user, we avoid inconsistencies in the votes casted. On top of that maps allow to rescind a vote, therefore also addressing (a). Effectively, it does improve over the previously presented solutions. We chose this presentation, since it is the best available option. In the following we will address issues of the implemented solution.

E. Problems of Implementation

We are happy that Maps resolve issuse that we might encounter on application level. However it prompts a new set of issues:

- We attempted using flags. As explained before, flags let True win over False. Deleting a flag to undo a vote is resolved on map level and thus through vector clocks. Therefore this solution has a mismatch in the mechanism which updates to different states of a vote. Further, we expected a user not being able to downvote a post after upvoting, yet the application works. We hope to investigate the inconsistency between documentation and actual results.
- We also used registers. Those rely on vector clocks to determine the correct value and can store any binary value. It matched our needs, since it efficiently represented the votes and had the flexibility required to change the vote. It also uses vector clocks to handle updates for all three states, therefore it is better to reason about than using flags.

Out of all possible options using maps with registers appears to be the best option. It provides advantages over all other options, but inherits one problem. As mentioned towards the end of IV-C.2, vector clocks bring a set issues. Specifically, conflict resolutions are pushed down to logical clocks. Effectively, we are versioning data, and using that system to resolve problems, thereby defeating some of the ideas of CRDTs.

V. Implementing Messaging

The second functionality we wanted to migrate is messaging. In Vataxia messaging is in the vain of emails. Instead of conversations or threads, each message is treated separately.

A. Implementation Options

We elected to implement messaging, since existing blog posts suggested that this is a primary use case for CRDTs. However, the blog posts did not disclose what data structure they used. Therefore again, we considered a range of implementation options:

1) Our initial impression was that a G-Set will be a good fit to store messages, since it is built to store unique items. In this case our set item must contain a timestamp, the message and the sender to be considered unique. However, we were not able to store that set on the user, since both conversation partners need to be able to see the message.
   To enable this, we created a map as a new data bucket. The keys of that map represent messages sent, while the values contained all information about the message. We designed they key to be a combination of the timestamp and the sender. In this context the assumption is that a sender will not send multiple messages within a second. Since we elected not to modify the frontend, we are taking the timestamps on the backend. If we were able to generate the timestamps on the frontend, we would be able to resend messages to the server in the case of communication failure without a duplicate message being stored. To reference messages in the map we maintained a G-Set per user that references each conversation they participate in. Note, that this supports group messages without a problem.
   A major hurdle we discovered was message ordering. A user should be presented with the messages in the order that he/she received them (latest first). A set data structure is not designed for ordering, instead it just maintains a set of values. Specifically Riak does not offer any inbuilt support. Therefore we resorted to application level ordering. This approach is problematic, because we need to load all message keys into memory and sort them. Especially at scale this generates significant overhead.
2) The other option we considered is to maintain a nested map. The key of the outer map is the receiver, while the key of the inner map is the sender. Within the second map messages are maintained as a G-Set. This allows us to lookup all received messages very quickly. It does however lack several key features:
   - To lookup all sent messages, the application has to traverse all possible senders.
   - Ordering is again achieved on application level.
   - Can not possibly support group conversa-

tions.

We decided on option 1, since it does offer some advantages over the alternative. We do acknowledge that this approach has some weaknesses:

- Vataxia supports the deletion of a message. The G-Set that maintains references to every conversation can not possibly support this. Our next best choice is to use a 2P set, so that a deletion essentially just removes the message from viewable messages. To delete the actual message, we can simply delete the key from the LWW map. We will further discuss issues caused by deletion in section VII-C.
- In the same vain, user deletion is complicated. It requires us to delete all messages individually, and then need to remove the reference to the message from other users.

As in section IV, we see that using CRDTs poses several problems. It is hard to represent the functionality we desire to implement with CRDTs.

VI. Industry Usage

To extend our analysis of CRDT usage in modern application development, we examined companies that successfully use CRDT database solutions.

A. Riot Games, League of Legends

Riot Games initially had their friends list data stored in MySQL, but found that the MySQL master node was a single-point of failure for their service and constrained their horizontal scaling capabilities [4]. They considered Cassandra, but decided that a schema-less solution would encourage faster iteration for their engineering teams, so they settled on Riak. League of Legends boasts over 100 million monthly active users, making Riot Games the owner of the likely the largest throughput CRDT system. Riot Games lets Riak store all conflicts in sibling lists and surfaces these conflicts to the application layer, which resolves them. They represent a friends list as a list of friends with a mutation log. If a player receives two friend requests from two different individuals at the same time, this could produce two different friends list that would be be merged at the application level by applying the operations in the mutation logs to reach convergence. In their blog post, details of the application layer conflict-resolution logic are sparse.

B. SoundCloud, Roshi

SoundCloud developed their own CRDT store called Roshi. Roshi is a time series event storage layer built on top of Redis and runs using CRDTs. Roshi is open-source and SoundCloud also published a technical blog post describing the motivation and high-level ideas behind Roshi [5]. They describe the underlying structure of Roshi as a modified LWW Set, with inline garbage collection, and currently use Roshi to back SoundCloud's stream, which is essentially an activity timeline.

However, they acknowledge that the obvious reaction to Roshi is to ask why we didn't implement it with an existing, proven data system like Cassandra. Their preemptive response to this question suggests that building on top of existing systems that are not built in-house has many costs: "costs like mapping your domain to the generic language of the system, learning the subtleties of the implementation, operating it at scale, and dealing with bugs that your likely novel use cases may reveal." They even suggest that the lack of ownership that their engineering team may feel if they use an existing standard data system is significant enough to highlight. We find these reasons weak and suggestive of the fact that SoundCloud developed Roshi as an interesting engineering challenge, to get the opportunity to play with CRDTs, but not because existing data systems were insufficient for their requirements. This weakens their usage of CRDTs as it is not clear what unique benefits Roshi provides for SoundCloud, as its development was not problem-driven, but rather engineering-driven.

VII. Problems of CRDTs

A. Centralization and Decentralization

CRDTs have the advantage over Operational Transform algorithms in that they do not require a centralized server to obtain eventual consistency. However, when we implement CRDTs in a database context, centralization is necessarily imposed, and we lose this unique advantage. We then lose the additional benefit of abstracting away

conflict resolution to the data level, because given the requirement of a centralized server, traditional techniques to achieve varying degrees of consistency are less constraining semantic-wise.

B. Opaque APIs

CRDTs are a complicated technical matter founded upon many years of mathematical research. Industry CRDT data stores such as Riak and Roshi are built based upon these research papers and ideas, but often times obscure the underlying semantic specifications of their implementations. Even with Riak's considerable documentation, the underlying mechanisms and details of their CRDT management and resolution often times are ambiguous or omitted. When an API abstracts away a complex underlying consistency mechanism, it is often dangerous to not have a transparent view into how the data layer operates as it inevitably leads to developer surprises. This is exacerbated by the fact that CRDTs, even at a theoretical level, are complex, asynchronous, and distributed, which makes maintaining a mental model of the operations challenging for engineers.

C. Privacy Concerns

During our work with CRDT, we started to understand the semantics of CRDTs more exactly. As we analyzed the semantics of messaging, we grew concerned with privacy and regulatory compliance. We identified two internal structures that will retain data outside of the queryable data itself:

- Log: CRDTs are fundamentally maintained through a log of changes to apply. This method is common, not only for CRDT stores, but also by many relational data stores. It is common to dispose the data once we know that all replicas are sufficiently up to date. As mentioned in section II-A, this is also possible for CRDTs. Therefore this is not a primary concern.
- Set: A major issue with sets is that deletion is not possible. A 2P-Set will only add a data entry to the tombstone set, however the data is not deleted from the store. Even though data is not queryable any more, it is still stored. Similarly, data can never be removed from a G-Set.

Any data required to comply with the EU "Right to be forgotten" can fundamentally not be stored as a CRDT-Set. Besides our concerns with the viability of CRDTs as a data structure itself, we see privacy as another major hurdle to adoption.

D. Manual Conflict Resolution

One of the largest appeals of CRDTs is their deterministic convergence which guarantees strong eventual consistency. In theory, this sidesteps the pain of using complex and expensive coordination protocols to achieve a similar semantic and leverages the mathematical properties of the idempotent and commutative operations. In practice, it is difficult to reap these benefits.

Riak allows developers to configure how conflict resolution happens [2]. One option is to use timestamp-based resolution, which can be enabled by setting the `allow_mult` and `last_write_wins` parameters to `false`. In timestamp-based resolution, Riak resolves all conflicts based on the timestamps assigned to each Riak object. As is common knowledge in the distributed systems world, using timestamps is insufficient. In the case of Riak, the documentation explicitly states that using timestamps may actually lead to data loss.

A second option is to use last-write wins resolution, which can be enabled by setting the `allow_mult` parameter to `false` and the `last_write_wins` parameter to `true`. Doing so will stop Riak from using internal conflict resolution and instead apply the latest write whenever a conflict arises. As a result, it is easy to see that concurrent writes will lead to some writes being lost. The documentation explicitly acknowledges this as well, sugggesting that LWW resolution only be used when concurrent updates are not possible.

The last option that Riak provides is application-level conflict resolution, which can be enabled by setting the `allow_mult` parameter to `true` and the `last_write_wins` parameter to `false`. This setting has Riak store all conflicts in the data itself and surface these options to the application layer, which should then apply some business logic to manually resolve the conflict. As of Riak version 2.0, this is the default setting for CRDT buckets and the recommended configuration. However, by

deferring the responsibility of CRDT conflict resolution to the application programmer, the CRDT abstraction becomes leaky and an enormous burden is placed on the developer. Writing domain-specific conflict resolution code is a non-trivial task as it requires careful enumeration of all conflict scenarios and appropriate handling of each scenario as to not violate data-level consistency guarantees as well as application-level consistency guarantees.

VIII. Use Cases for CRDTs

While our project is focused on examining the practicality of using CRDTs for general application development, there has been more success using these data types for specific types of applications: namely collaborative editing, offline workflows, and peer-to-peer. In fact, the Atom text editor recently launched a real-time collaborative coding feature powered by CRDTs [6]. We believe peer-to-peer applications are also well-suited for CRDTs because the inherent architecture of the network leverages the decentralization benefit of using CRDTs, and fares better than other P2P data model alternatives.

IX. Conclusions

CRDTs propose an exciting concept for data storage in distributed systems. Our experiments however show that the CRDTs that are currently available do not offer sufficient semantics to be used in most applications. We have shown that CRDT are not able to represent many features in a sensible fashion. Instead one often ends up achieving consistency through vector clocks, thereby diminish the advantages of CRDTs. On top of that CRDTs can pose significant challenges in terms of privacy. Overall, CRDTs do not appear to work for databases in practice. We are excited about the computational model that CRDTs propose and hope that future research can make CRDTs more viable as a general purpose datatype.

X. Future Work

In the future, we believe that examining existing open-source implementations of CRDTs for peer-to-peer application will reveal more non-trivial benefits of the data type. CRDTs can be an innovative solution for update propagation in distributed systems. However, it can only apply to cases where the task can be expressed in CRDT semantics. Future research needs to identify a broader set of operations to make CRDTs a feasible data type for most use cases.

References

[1] Marc Shapiro, Nuno Preguia, Carlos Baquero, Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. [Research Report] RR-7506, Inria Centre ParisRocquencourt; INRIA. 2011, pp.50. <inria-00555588>

[2] Basho Technologies. Riak-KV Documentation. http://basho.com/products/riak-kv/

[3] Martin Kleppmann, Alastai R. Beresford. A Conflict-Free Replicated JSON Datatype. [cs.DC] 15 Aug 2017 ¡arXiv:1608.03960v3¿

[4] Riot Games Engineering Blog. Chat Service Architecture: Persistence. 13 Jan 2016. https://engineering.riotgames.com/news/chat-service-architecture-persistence

[5] SoundCloud Developers Blog. Roshi: a CRDT system for timestamped events. 9 May 2014. https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events

[6] Atom Blog. Code together in real time with Teletype for Atom. 15 November 2017. https://blog.atom.io/2017/11/15/code-together-in-real-time-with-teletype-for-atom.html