CRDTree

Braxton Hall
Haotian Yang
Ray Zhang
CPSC 538B 20W1: Distributed Systems Abstractions

1. Introduction

Collaborative applications allow multiple users to edit shared data, and have become a necessity in both industry and academia. Systems like Google Docs and Git both provide means of either real-time or asynchronous collaboration. We aim to build an available, peer-to-peer system which supports both real-time and asynchronous collaboration.

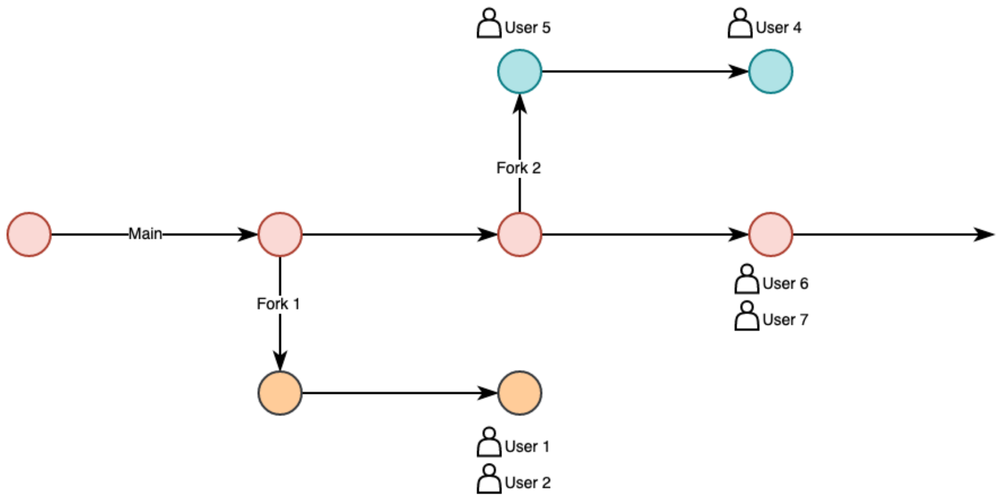

Figure1. The overall conceptual view of CRDTree from user's perspective

2. Background

2.1 Scenario: Collaborative Proposals

A document editing system allows a team to work together on a document, each with their own editor. This may be a paper or even source code. Users are able to see changes in real time. At some point, there may be some disagreement on how to proceed on a subcomponent of the document. Users split into subteams, each with their own version of the document, where they may continue to collaborate in real time with their subteam. Eventually, the subteams reconvene to choose the "correct" subcomponent, and continue working from that version of the document. A subteam is also eligible to be split into subteams as competing revisions may occur *for the subteams*.

If a mistake is found in the document, users should be able to search the history for when the bad revision was created. Thus all intermediate changes must be retained from collaboration on proposals in the final document.

2.2 Requirements

1. Real time collaboration on shared data
2. Asynchronous workflows
    a. Real time collaboration on data change proposals
    b. Data change proposals for proposals
3. Retain intermediate changes from working drafts of proposals
4. Storage usage should be minimal

2.3 Existing Solutions

Google Docs supports real-time editing, however clients do not collaborate with each other, but rather with a central server. This creates a worrisome truck factor, as there is a single point of failure. This creates the requirement for a peer-to-peer system, with support for data lookup and retrieval.

A Conflict-free Replicated Data Type (CRDT) supports peer-to-peer real-time collaboration through its guarantee of strong eventual consistency. When dealing with collaborative operations with multiple users, a typical protocol involving a CRDT performs all the users' operations eagerly, merging data as soon as possible. However, one may not wish to perform these merges as soon as they receive operations from other users. Asynchronous collaboration becomes handy in the case where an owner of data may wish to view how changes affect it, and preview modifications before accepting them as a merge or even discarding them.

Simply creating a peer-to-peer Google Docs complete with change proposals, still missing is the ability to collaborate on a proposal, or even make a proposal *for* a proposal. These kinds of workflows are supported by Git.

Additionally, Git commits are made entirely manually, rather than real time. Merging changes often creates merge conflicts which must be manually resolved. A CRDT would absolve users from having to engage with merge conflicts, and work on shared data in real-time.

On top of Git, users are able to use an additional application like Visual Studio Live Share or JetBrains' Code With Me to collaborate on a branch before requesting that it be merged. This satisfies the requirements of real time *and* asynchronous workflows. However, intermediate changes from the external real-time editing application are lost upon creating a commit and merge to the main branch. Additionally, while some history is saved in the application underneath, Code With Me only tracks a single flat history, which causes conflict when concurrent users attempt to undo their own edits.

3. Approach

In this section we outline our approach for CRDTree, a library for a Conflict Free Replicated Data Type with the extension of forks and joins. We use the names fork and join, not only to avoid the naming collision with “merge” in CRDT literature, but also because to a user, there are no single commits, just a stream of changes.

3.1 Design

3.1.1 Assumptions

- Messages will eventually be delivered if the recipient comes online
- Messages will not be corrupted
- Messages may be delivered out of order
- There are no Byzantine faults (processes are only fail-stop)
- Nodes may go offline indefinitely, or recover
- New nodes may be join the system

3.1.2 CRDTree

Branches should support intentionally letting states diverge. Creating a branch should divert state, letting different sets of collaborators work in partitions each with their own shared and eventually consistent state.

Joins should allow for the dissolution of these partitions. In a join of branch A to branch B, all operations from within the partition of collaborators for branch A should eventually be propagated to the partition of collaborators for branch B where the join was accepted. If branch B has already been merged into another branch C, then those changes should also eventually be propagated to that corresponding partition of collaborators as well.

In a perfect network collaborating on a basic operation based CRDT object o, every process p_i containing a replica o_i broadcasts o_i’s operations s infinitely often, allowing infinitely many merge operations m_j on every process p_j. Under the assumption that every message is *eventually* delivered, this presupposition is a direct proof of Eventual Consistency as described in [5].

However in practice, merges may be delayed by a network partition (created by interference or outages). Within one partition, state may accumulate, and then be delivered all at once when the network partitioning has ceased.

Conceptually, our notion of CRDT forking resembles this temporary partitioning. An artificial and user-controlled partition allows a set of users to collaborate on data that is not automatically merged back to all other replicas. Their cumulative changes may be previewed and even approved before the partition is removed.

Changes that are delivered for one branch on a process should not be reflected in the data visible to applications on another, even though the two branches co-exist in the same process. This means that a process, when creating a branch, creates an additional replica with the same artificial partitioning between the originating branch and the new branch.

Creating a branch from a parent CRDT object o^y may then be seen as creating a new CRDT object o^x with some amount of history shared with o^y, which can be independently collaborated on as if creating an artificial network partition.

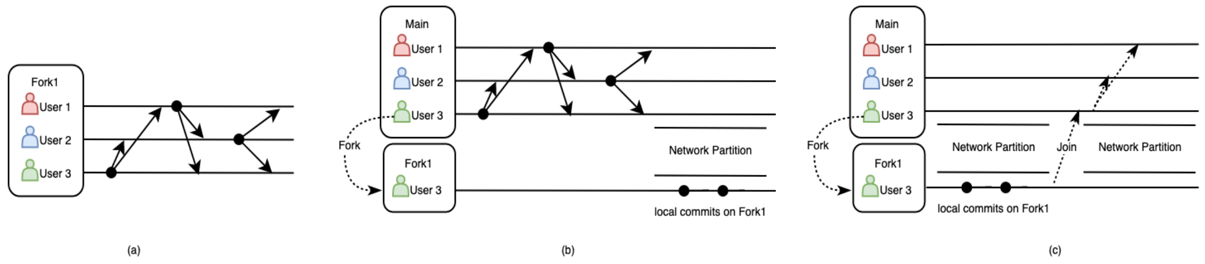

Figure 2.(a) CRDT Collaboration (b) CRDTree Fork Operation (c) CRDTree Join Operation

A single branch retains all the local properties of a CRDT, and may be seen as a CRDT collaborated on through a subnetwork within the entire CRDTree network. Additionally, they may also be joined into each other, provided there is some common ancestor to both the joiner and the joinee. As branches begin as clones of a CRDT object, we know that a branch must have the same initial state s^0 and domain S. Thus all branches o^x of o^y extend *replicas* of o^y, and *may* be merged.

Accepting a join conceptually resembles dissolving an artificial network partition between replicas in a single direction, immediately followed by restoring it. Thus it follows that, if all branches {o^0,...o^n} are joined into some default fork replica o^r (i.e.: all network partitions dissolved) and no other changes are made on {o^0,...o^n}, then we retain the same eventual consistency and strong eventual consistency guarantees as a regular CRDT on o^r.

Forks and joins are explicitly stored in branches' histories as operations. Any single branch has all the properties of a CRDT, with its own set of collaborators. It may be seen as a CRDT collaborated on through a subnetwork of the full CRDTree network.

3.1.2.1 Storage Reduction

The naive solution of forking by creating a copy of a CRDT and redelivering operations upon join causes every fork to duplicate the space usage on a process, which fails to meet the requirement that storage usage should be minimal. In order to mitigate storage usage, instead of copying history, the "initial operation" of a fork will be the fork operation, which acts as a "pointer" to where more history can be found.

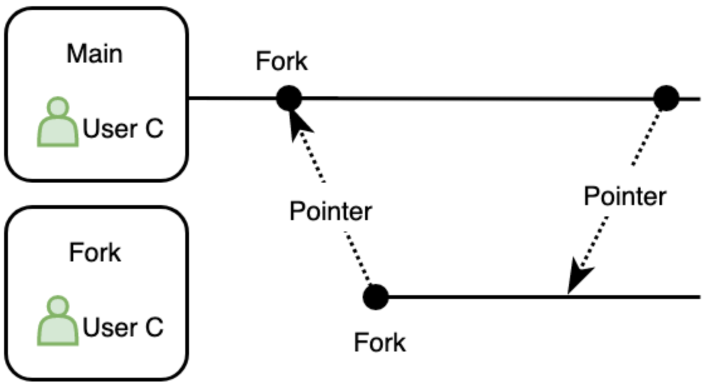

Figure 3. The initial state of the fork is a pointer to more history, as is a join operation

A branch o^x is defined by its

1. name,
2. initial operation u_x0, which lies on the history of its parent o^y
3. history of new operations u_xi following u_x0, local to o^x

To query the extended history of o^x before its initial operation u_x0, the query must go through the operation history tracked by its parent o^y.

The root of a CRDTree has no parent or initial operation.

3.1.2.1 Deleting a Branch

Say there exists some branch known by two processes. If process p_j is offline, and process p_i were to delete their replica o_ix, there would be no node to retrieve missing operations from when process p_j returns online. For this reason, we do not intend to support deleting a replica from a process.

3.1.3 The CRDTree Protocol

Collaborating on a basic CRDT may be as simple as broadcasting all operations to all nodes in the network. For CRDTree, this simple protocol would also be sufficient. However, it comes at the cost of sending messages to processes who may not be interested. These messages also contain operations which must be stored. Operations on a branch which a process is not following only become relevant to that process if the remote branch has been joined into one of the local branches.

To save both storage and bandwidth for unjoined operations to some local branch, we define the CRDTree Protocol, which realises the artificial network partition when possible.

In addition to the data CRDTree which users interact with to store their data, the protocol requires a second metadata CRDT. The metadata CRDT is shared by all processes, and consists of a set of named sets. Names correspond to branch names that exist on any process in the network. The named set for branch o^x contains identifiers for all processes following o^x, as well as any other branches o^z which o^x has at some point joined into, paired with the latest join operation from o^x into o^z.

Using this metadata CRDT, and an operation for a local branch, one may recursively compute all other processes who must learn about this operation, and broadcast the operation only to that subset of the network. A tentative elaboration of this metadata’s structure can be seen in Section 6.

Joins, forks, and checkouts of a branch all manipulate this shared data. As we do not support deleting a fork, items will never be removed from any set or nested set in this shared metadata CRDT.

3.2 Implementation

3.2.1 CRDTree Implementation

We seek to build CRDTree from the ground up in TypeScript.

3.2.2 CRDTree Protocol Library Implementation

As described in 3.1.3, whenever there is an operation u_xi on a branch o^x, our space-saving protocol initiates a recursive message propagation to the users who are effectively subscribed to o^x at the point u_xi falls in its history. This protocol requires each user to connect to all other users in the network and send messages to an arbitrary subset of users.

In a peer-to-peer network, each peer needs to discover, identify, locate, and create a connection with other nodes. Building the peer-to-peer network is the crucial foundation of the CRDTree Protocol Library, and instead of building it from scratch, we seek to leverage the libp2p [7] library in Golang.

4. Evaluation

4.1 Research Questions

- RQ1: Is our implemented CRDTree correct?
- RQ2: Is our CRDTree performant?
- RQ3: Is our CRDTree and accompanying Protocol Library usable?
- RQ4: How does CRDTree compare to existing solutions?

4.2 Methodology

To test RQ1 and ensure that our implementation matches the specified merge and branch semantics (with out-of-order message delivery), we will construct an automated test suite, with programmatically withheld and delivered messages simulating out-of-order message delivery.

Additionally, we will build a lightweight web application that supports document editing with integrated branch and pull request workflows. We will then use this application over the internet, which will answer RQ2.

Finally, we will measure the local storage as well as the commit latency of CRDTree to compare it with GitHub, Google Docs and peer-to-peer Git, answering RQ[2,4].

5. Timeline

Milestone 1: Environment Setup, API Design, Testing (Oct 29)

We will use this development period for further elicit the finer requirements of our implementation, as well as create our shared repository

Requirements

- FR1: Should have a command line invokable test suite
- FR2: Should have signature and stubs for the high level API
    - For CRDTree this includes merges, updates, network integration endpoints, forks and joins
    - For the CRDTree Protocol Library this includes connecting, as well as persisting the local CRDTree
- NFR1: Should have a comprehensive test suite for CRDT as well as the CRDTree extension. Obvious priorities include:
    - Should test that concurrent operations are deterministically merged
    - Should test that state reflects the history of operations
    - Should test that forks reflect the history of the tree they forked from
    - Should test that state reflects the history of operations that occurred on forks joined into the current fork
- NRF2: Should have a partial test suite for the CRDTree Protocol Library

FR[1,2] may be manually evaluated.

Milestone 2: CRDT Implementation (Nov 15)

This development period will be used to build a working CRDT that renders JSON

Requirements

- FR1: Should be able to initialize a CRDT
- FR2: Should be able to merge two CRDT given their initial states are the same
- FR3: Should be able to render a CRDTree
- NFR1: Merges should terminate in a reasonable amount of time, less than a second
- NFR2: Memory allocation should scale at worst polynomially in operations

FR[1-3] will be evaluated using our automated test suite created for Milestone 1.

Milestone 3: CRDTree Extension (Nov 29)

- FR1: Should be able to make a fork
- FR2: Should be able to join a fork
- FR3: Should be able to render a history that contains forks or joins

FR[1-3] will be evaluated using our automated test suite created for Milestone 1.

“Milestone 4”: CRDTree Protocol Library and Evaluation (Dec 21)

We have chosen to call the final report delivery “Milestone 4,” as we will be using it to deliver further implementation. In addition to the CRDTree Protocol Library for this development period, we will deliver our final reports

- FR1: Should be able to connect to other users
- FR2: Should be able to calculate processes that are subscribed to an operation
- FR3: Should send new operations to processes on the same fork
- FR4: Should not send new operations to processes not on the same fork
- NFR1: Should have a complete test suite for the CRDTree Protocol Library

FR[1-3] will be evaluated using the extended automated test suite created for “Milestone 4.”

6. Appendix

Tentative CRDTree API

```
type CRDTreeTransport<T> = unknown; // used for sending updates across the network
type ID = unknown;

interface CRDTree<T> {
   new (intialState: T): CRDTree<T>;
   new (from: CRDTreeTransport<T>): CRDTree<T>;

   render(): T;

   clone(): CRDTreeTransport<T>; // for when a new node joins the network
   // returns affected forks, throws DifferentForkException
   merge(remote: CRDTree<T> | CRDTreeTransport<T>): ID[];
   update(state: Partial<T>): void; // To delete something, set it to undefined
   onUpdate(callback: (update: CRDTreeTransport<T>) => void);

   ref(): ID;
   listRefs(): ID[];
   fork(): ID;
   join(ref: ID): void; // throws UnrelatedHistoryException (same fork is a no-op)
   checkout(ref: ID): void;
}
```

Tentative CRDTree Protocol Metadata Definition

```
interface SubscriberData {
   [name: Branch]: {
       processes: Set<Process>
```

```
        joinedIntoBranches: {[name: Branch]: Operation}
    };
}

function getSubscribers(data: SubscriberData, branch: Branch, operation:
Operation): Set<Process> {
    const {processes, joinedIntoBranches} = data[branch];
    const subscribers = new Set(processes);
    for (const joinedIntoBranch, joinPoint in joinedIntoBranches) {
        if (operation.happensBefore(joinPoint)) {
            subscribers.addAll(getSubscribers(data, joinedIntoBranch, operation));
        }
    }
    return subscribers;
}
```

7. References

- [2] “CRDTs and the Quest for Distributed Consistency.” Web.
  CRDTs and the Quest for Distributed Consistency
- [3] Jagadeesan R., Riely J. (2018) Eventual Consistency for CRDTs. In: Ahmed A. (eds) Programming Languages and Systems. ESOP 2018. Lecture Notes in Computer Science, vol 10801. Springer, Cham. https://doi.org/10.1007/978-3-319-89884-1_34
- [4] Preguiça, Nuno. "Conflict-free replicated data types: An overview." arXiv preprint arXiv:1806.10254 (2018).
- [5] Shapiro, Marc, et al. "Conflict-free replicated data types." Symposium on Self-Stabilizing Systems. Springer, Berlin, Heidelberg, 2011.
- [6] Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31.4 (2001): 149-160.
- [7] libp2p https://libp2p.io/