CONSTRUCTOR UNIVERSITY

Local-first Collaboration on Relational Data

by

Veronika Sirotkina

Bachelor Thesis in Computer Science

Submission: April 19, 2024 Supervisor: ??? Anton Podkopaev

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Sirotkina, Veronika
Matriculation number	30006541
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

.....

Date, Signature

Abstract

TODO

Contents

1 Introduction

With internet becoming more accessible and cloud-based software gaining momentum, more apps started to heavily depend on the presence of internet connection to work properly. This is especially true for apps that need to synchronize clients' data across multiple devices or apps that allow for real-time collaboration between clients.

The most obvious and maybe the easiest way to enable synchronization across devices is naturally to store an authoritative copy of the data in the cloud and provide a centralized server to manage changes coming from the clients. Real-time collaboration can be achieved by implementing operational transformation algorithms [12].

This approach is, of course, viable and is widely used in practice, but it has a big downside. Without internet connection access to the data is either lost completely or it is only available in a read-only mode. The access can also be lost if the provider company's servers are down, or if the company stops supporting the product, or if the company finds the contents of the document inappropriate [2].

Local-first software [8] is a new approach to designing collaborative software. In the context of this paper the most notable feature of local-first software is that it allows clients to edit shared documents even when they are offline. The changes can be integrated with the upstream version once the connectivity is reestablished.

GanttProject [5, 4] is an open-source tool for building Gantt diagrams. It's originally local-only, but at the current time it also provides a cloud storage [3]. A work-in-progress module called Colloboque will enable real-time colloboration in GanttProject.

The goal of this project is to enable local-first real-time collaboration in GanttProject by further developing Colloboque.

2 Background

TODO

2.1 Operational Transformation

Operational Transformation [12] or OT is a technology for managing concurrent updates, particularly in collaboration software. OT algorithms assume that there are multiple clients. All clients have a replica of the same document and they can modify the document with a predetermined set of operations independently from each other.

When a client recieves an update from another client, he doesn't apply the recieved operation immediately. He compares the incoming change against other observed concurrent changes, transforms the operation and then applies it. The transformation algorithm must ensure that the states of all clients converge.

2.1.1 OT for plain text

TODO: PICTURE

In this section I will present an example of how OT can be used for managing concurrent insertions in plain text documents.

There are two clients, Alice and Bob, and each of them has a copy of a document. The document contains a single sentence: ”Hello World”. The only defined operation is an insertion of a single character at a specified position: `insert(char, pos)`.

Alice inserts a comma after ”Hello” with `insert(',', 5)` and gets ”Hello, World”, while Bob inserts an exclamation mark after ”World” with `insert('!', 11)` and gets ”Hello World!”. After that they exchange the performed operations.

If Alice and Bob both apply each other's changes as is, Alice will get ”Hello, Worl!d”, and Bob will get ”Hello, World!” - the two documents now diverge. Instead, when Alice gets `insert('!', 11)` she compares it with the operation she performed concurrently `insert(',', 5)`, sees that the indexing has shifted and transforms `insert('!', 11)` into `insert('!', 12)`. After applying this transformed operation she will get ”Hello, World!”.

Bob performs similar actions to learn that he doesn't need to transform Alice's operation, applies it as is and also gets ”Hello, World!”.

2.1.2 Google Wave

Google Wave [6, 7] was a communication platform developed by Google using Operational Transformation approach. It was launched in September 2009. Google Wave combined elements of email, instant messaging, wikis, and social networks into a single platform.

Google Wave had features such as real-time collaboration, allowing users to work together on documents known as ”waves” by typing messages, inserting images, and adding other content. It supported the embedding of media objects like videos, maps, and polls directly into waves.

Despite its innovative features, Google Wave failed to gain popularity. Google discontinued Google Wave in August 2010 due to low user adoption, but its technology and concepts influenced subsequent Google products like Google Docs and Google Drive.

2.2 Local-first Software

Local-first software was first formalized in 2019. The original paper [8] suggests 7 principles of local-first software. Here I will list 4 which I find the most relevant.

- **Synchronization across devices**
  Imagine a client who uses a document editor on multiple devices. Synchronization involves tracking changes made by the client on one device and ensuring these changes are reflected across all other devices used by the client. In the end data on all devices must reach the same state. This is a very convenient feature that lets clients acces their work from any device.
- **The network is optional**
  Nowadays it is normal for many apps to lose most of their functionality if internet connection is unstable. TODO: EXAMPLES. For local-first software it is important that it should retain its core functionality even when the device is offline. The client is still able to view and edit documents as he pleases, and the changes made while being offline are integrated with other replicas when the device connects to the network again.

- **Seamless collaboration**
  Real-time collaboration is a very attractive feature that lets multiplea people work on a single document simultaneously. Some notable examples of web-apps that allow for real-time collaboration are Google Docs, Google Sheets, Figma, etc. Usually such apps don't allow to edit documents offline with some Google products like Google Docs being an exception. For local-first software the aim is to provide collaboration functionality on par with cloud-based apps like Figma while retaining optionality of the network.
- **Ultimate ownership and control**
  As a popular saying goes, there is no cloud - it's just someone else's computer. When user data is stored in the cloud, the ultimate ownership of the data belongs to the corporation that provides the software. The user might lose acces to their data because of technical problems, or, what's more concerning, the company might bar the user from accessing their data on a whim. For example, by introducing a paid subscription or banning the user because their data presumably violates the service's policies [2]. With local-first software, the data physically belongs to the user and the service provider can't take the user's access.

2.3 CRDT

CRDT stands for Conflict-Free Replicated Data Type [11]. It's a type of data structure designed for distributed systems. Objects of CRDTs can be replicated across multiple nodes, and each node can independently update the object. When information about the updates or, depending on the flavour of a particular CRDT specification, the state of each node is shared, CRDTs guarantee that all replicas will eventually reach the same state.

CRDT object has an identifier, physical content and an initial state. Any two objects with the same identifiers are called replicas of each other. CRDT specification defines a set of update operations, as well as read-only queries.

A CRDT can implement any logical type like Counter, Set, List, etc. The state of a CRDT's logical type is represented by its abstract state. For example, a Counter CRDT can be implemented as a tuple of values, where each replica can only change its designated value. The abstract state of such CRDT is the sum of all values in a tuple.

TODO: CLARIFY COUNTER CRDT?

CRDT guarantees that all well-formed update operations are commutative and idempotent with respect to the abstract state of an object. Because of these properties, replicas can make updates independently, without communicating with other replicas. As long as all updates are eventually communicated, all replicas are guaranteed to end up with the same abstract state.

CRDT types can be further categorized into two groups: state-based CRDTs and operation-based CRDTs. It is worth noting that state-based CRDTs can always be emulated by operation-based CRDTs and operation-based CRDTs can always be emulated by state-based CRDTs. But for some structures it's simpler to think in terms of state-based CRDT, while some feel more natural as an operation-based CRDT.

2.3.1 State-based CRDT

For state-based CRDT one has to specify a merge function, which will merge together states of two different replicas. Requirements for the merge function can be formulated in terms of causal history.

Definition 2.1 (Causal History — state-based)**.** For any replica x_i of x:

- Initially, $C(x_i) = \emptyset$.
- After executing update operation f, $C(f(x_i)) = C(x_i) \cup \{f\}$.
- After executing merge against states x_i, x_j, $C(merge(x_i, x_j)) = C(x_i) \cup C(x_j)$.

Here is an example implementation of a state-based Counter CRDT

```
Counter {
    // initial state
    val counter = (0, 0, 0)

    // identifier from {0, 1, 2}
    val id

    // update operation
    fun increment() {
        counter[id] += 1
    }

    // query
    fun value() = sum(counter)

    // merge function
    fun merge(c1, c2) {
        merged = (0, 0, 0)
        for (i in 0..2) {
            merged[i] = max(c1[i], c2[i])
        }
        return merged
    }
}
```

2.3.2 Operation-based CRDT

TODO

Definition 2.2 (Causal History — operation-based)**.** For any replica x_i of x:

- Initially, $C(x_i) = \emptyset$.
- After executing update operation f, $C(f(x_i)) = C(x_i) \cup \{f\}$.

2.3.3 Automerge

Automerge [1] is a library that provides a JSON-like CRDT for JavaScript. It was designed specifically for implementing local-first software. Automerge documents support all JSON primitive data types, as well as Map, List, Text, and Counter.

TODO: AUTOMERGE USAGE EXAMPLE

Automerge documents support both merging full documents and applying individual changes. To define rules for merging documents, it's enough to define rules for merging Maps, Lists, Text and Counters. Full explanation of the merging rules in Automerge can be found in Automerge documentation [9].

Some conflicts can't be automaticaly resolved, for example if two replicas concurrently set the same key. In events like this Automerge produces a Conflict object, which contains updates from both replicas. They can be used to allow the user to pick the value manually, but this is not mandatory. Conflict objects can be completely ignored, in which case a single value is picked automaticaly from the list of available options presented by the Conflict object. The library guarantees that if two replicas encounter the same Conflict, it will be resolved in a consistent way, ensuring that the two replicas still converge.

The core of Automerge is written in Rust. Consider the following example using Automerge's Rust API demonstrating Conflict objects.

TODO: FIX LISTING

```
use automerge::ReadDoc;
use automerge::transaction::Transactable;

fn main() {
    let mut first_doc = automerge::Automerge::new();
    let mut second_doc = first_doc.fork();
    first_doc.transact::<_, _, automerge::AutomergeError>(|doc| {
        Ok(doc.put(automerge::ROOT, "key", "first value").unwrap())
    }).unwrap();
    second_doc.transact::<_, _, automerge::AutomergeError>(|doc| {
        Ok(doc.put(automerge::ROOT, "key", "second value").unwrap())
    }).unwrap();
    first_doc.merge(&mut second_doc).unwrap();
    second_doc.merge(&mut first_doc).unwrap();
    let conflict1: Vec<_> = first_doc.get_all(automerge::ROOT, "key").unwrap()
    let value1 = first_doc.get(automerge::ROOT, "key").unwrap().unwrap().0;
    let conflict2: Vec<_> = second_doc.get_all(automerge::ROOT, "key").unwrap(
    let value2 = second_doc.get(automerge::ROOT, "key").unwrap().unwrap().0;

    println!("Conflict in the first document:\n{:?}", conflict1);
    println!("Conflict in the second document:\n{:?}", conflict2);
    println!("First conflict resolves to  {:?}", value1);
    println!("Second conflict resolves to {:?}", value2);

    // Possible output
    // Conflict in the first document:
    // [(Scalar(Str("first value")), Id(1, ActorID("1baddedabae54a6ca7a8aaf03d
```

```
    // Conflict in the second document:
    // [(Scalar(Str("first value")), Id(1, ActorID("1baddedabae54a6ca7a8aaf03d
    // First conflict resolves to Scalar(Str("second value"))
    // Second conflict resolves to Scalar(Str("second value"))

}
```

2.3.4 CRDT for Relational Data

Some work on designing CRDT for relational data have been done before [13]. In the paper the authors present a two layered architecture. The first layer is the Application Relation layer, or AR layer. The application can interact with the application layer as if it was a normal database. The second layer, called Conflict-free Replicated Relation layer or CRR layer, handles the logic of applying and propagating updates. The CRR layer contains an AR layer schema, augmented with additional data to implement CRDT logic.

Read-only queries can be executed directly on the application layer. Update requests are applied atomicaly to both AR and CRR schemas. Later they are propagated to other replicas.

The CRR layer is responsible for handling conflicts between local updates and concurrent updates propagated by other replicas. To achieve that the CRR layer stores rows in a special Set CRDT called CLSet, and all row attributes are stored in a Last-Writer-Wins register CRDT. Some numeric attributes are represented by Counter CRDT.

CLSet maintains a counter for each object in the set. This number is called the *causal length* of an object. Successful inserts and removals increment causal length, while unsuccessful operations have no effect. For example, removing an object that doesn't belong to the set has no effect, while removing an object that does belong to the set increments the object's causal length. Following this specification, we can conclude that an object belongs to the set if and only if its causal length is odd. When handling conflicting updates, an update with the greatest causal length wins.

Another problem that the CRR layer needs to solve is maintaining integrity constraints. If some constraints are violated when applying a local update the update can be simply rolled back. The main challenge comes from integrity constaint violations happening when merging updates from other replicas. In this case the CRR layer generates a new update that undoes the conflicting update and propagates it to other replicas.

The paper discusses three kinds of integrity constraints violations.

The first one is uniqueness constraint violation. When two updates conflict because of a uniqueness constraint violation, update with a higher timestamp is reverted.

The second one is referential integrity constraint violation. It can happen when one replica updates or inserts a row with a foreign key f, while another replica deletes the row containing f from the referenced table. In that case the update/insert operation is reverted, because the delete operation might have been the result of undoing an insert after uniqueness constraint violation. In this case undoing the delete operation will lead to encountering a uniqueness constraint violation again.

The third one is numeric constraint violation like lower and upper bounds. Since some numeric attributes behave like counters with additive updates, it's possible to get overflow

or underflow after merging concurrent updates. In this case the more recent updates are reverted.

The approach with issuing undo updates is not fit for all cases. For example some operations might have further side-effects, like sending an email or charging a bank account. An undo operation can't revert the impact made on the external world. Some papers TODO: CITE suggest that this problem could be solved by introducing quotas for certain operation. For example, if the account balance is 1000, and a client has 10 devices, each device can be issued a quota to deduct 100 from the account. If a device wants to deduct more, it must contact other devices to redistribute the quotas.

3 Approaches

In this section I will describe approaches considered during the course of my work.

3.1 CRDT approach

The first approach to enabling local-first collaboration in GanttProject is to implement a CRDT for relational data. With a robust relational data CRDT, the task would be very straightforward. Each client can store a replica of a CRDT object and propagate its updates to other clients either through a centralized server or a peer-to-peer connection.

The main challenge comes from implementing a relational data CRDT. The paper referenced in Section 2.3.4 gives a general direction on how such a CRDT can be implemented, but this is still no easy task. It's also unclear whether additional problems could arise due to the limited support of SQL features that the described CRDT provides. For example, it seems impossible to implement an auto-increment attribute. When there is a need to generate unique ids for newly inserted rows the authors of the paper generate random ids instead. There are probably many more such examples, but listing them requires further analysis.

The described relational data CRDT is implemented in a library for Elixir called Crecto. It can't be used easily since GanttProject is implemented in Java and Kotlin.

Taking into account the amount of work required to employ this approach and the assiciated concerns it was decided to think of other approaches which would be easier to implement and integrate with the existing codebase.

3.2 Time Warp approach

In distributed systems, "time warp" refers to a technique used to handle inconsistencies in the ordering of events across multiple nodes. In such systems, events may occur concurrently at different nodes. Due to network partitions and communication delays, the order in which events are observed can be different at different nodes.

Time warp allows a node to rewind its state to apply an out-of-order event. After that the node can also reapply the events that were undone during the rewind stage. TODO: PICTURE

This technique can be used to allow local-first collaboration with the help of a centralized server. Assume that the clients are able to perform time warp. The centralized server

receives update messages from clients and writes them to an append-only log. Let clients adhere to the following algorithm when producing updates:

1. Add an update to the local log
2. Notify the server about the update
3. Wait for the server to process the update
4. Recieve response from the server that the update was successfully committed
5. Mark the update as committed

Alternatively if the client went out of sync:

1. Add an update to the local log
2. Notify the server about the update
3. Wait for the server to process the update
4. Recieve response from the server that the update is out of sync. The server also sends the updates that must be applied by this client to sync with the server.
5. Client uses time warp to undo local updates and apply sync updates.
6. If local updates produce no conflicts, they are applied again, otherwise the user can be prompted to resolve the conflicts. The original updates are discarded, and if the user resolves the conflict, new updates are generated.

Notice that even if the update that the server received is out of sync, it can still be committed if it produces no conflicts. When the server send the client a list of sync updates, it can also inform the client that the out of sync update was committed successfuly after the sync updates. Then the client can also apply this local update after applying all the sync updates - there is no need to discard it.

To employ this approach, it is crucial that the client is able to revert recent updates. Unfortunately, this functionality doesn't come built-in with relational databases. To allow time warp on a database, one would have to be able to generate undo operations for all possible transactions. TODO: THIS IS PROBABLY NOT TRIVIAL, EXPLAIN WHY. COULDN'T FIND ANY ARTICLES.

TODO: DATA BRANCHING

3.3 Transaction replay approach

TODO: POLISH CODE LISTINGS

The Colloboque server stores a history of committed Postgres database transactions and the state of the project after each transaction. When a client loses connection to the server, he keeps commiting transactions locally. When connection is re-established, the set of changes applied by the client is sent to the server. The server then takes the state of the project that the out-of-sync client based its changes on and uses it to start two transactions. The first transaction contains all the changes that the client applied locally, while the second transaction contains all the changes processed by the server while the client was disconnected.

3.3.1 Postgres transaction isolation levels

In the described approach transactions have to make use of isolation level REPEATABLE READ, which is stronger than the default transaction isolation level.

The default mode for transactions in Postgres is READ COMMITTED [10]. It means that a transaction is allowed to read changes that where committed by other transactions after the current transaction has started.

```
BEGIN;
SELECT * FROM data; -- 0 rows
-- another transaction commits inserting a row
SELECT * FROM data; -- 1 rows
COMMIT;
```

When using REPEATABLE READ transaction mode, the changes from newly committed transactions are not visible, but if the current transaction makes a conflicting update, it will lead to a serialization error.

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM data; -- 1 rows
-- another transaction commits deleting a row
SELECT * FROM data; -- 1 rows
UPDATE data SET col = 1;
-- ERROR:  could not serialize access due to concurrent delete
```

With this approach the Postgres database can reliably identify whether the changes that the client made offline produce confilcts. If there are conflicts, the server discards the changes. The client will then have to sync with the server. An out-of-sync version of the document can be saved as a copy so that the user can compare the two versions manually.

TODO: DATA BRANCHING

4 Implementation

References

[1] *Automerge*. https://github.com/automerge/automerge.

[2] Brian Fung. “A mysterious message is locking Google Docs users out of their files”. In: *The Washington Post* (2017). URL: https://www.washingtonpost.com/news/the-switch/wp/2017/10/31/a-mysterious-message-is-locking-google-docs-users-out-of-their-files/.

[3] *GanttProject Cloud official web-page*. https://ganttproject.cloud/.

[4] *GanttProject GitHub repository*. https://github.com/bardsoftware/ganttproject.

[5] *GanttProject official web-page*. https://www.ganttproject.biz/.

[6] *Google Wave on Wikipedia*. https://en.wikipedia.org/wiki/Google_Wave.

[7] *Google Wave Overview*. https://youtu.be/p6pgxLaDdQw?si=QcI7beHUIwMFTo2o. 2009.

[8] Martin Kleppmann et al. “Local-first software: you own your data, in spite of the cloud”. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178. ISBN: 9781450369954. DOI: 10.1145/3359591.3359737. URL: https://doi.org/10.1145/3359591.3359737.

[9] *Merging rules for objects in Automerge documents*. https://automerge.org/docs/under-the-hood/merge_rules/.

[10] *PostgreSQL Transaction Isolation Levels documentation*. https://www.postgresql.org/docs/current/transaction-iso.html.

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

[12] Chengzheng Sun and Clarence Ellis. “Operational transformation in real-time group editors: issues, algorithms, and achievements”. In: *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. CSCW ’98. Seattle, Washington, USA: Association for Computing Machinery, 1998, pp. 59–68. ISBN: 1581130090. DOI: 10.1145/289444.289469. URL: https://doi.org/10.1145/289444.289469.

[13] Weihai Yu and Claudia-Lavinia Ignat. “Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge”. In: *IEEE International Conference on Smart Data Services, 2020 IEEE World Congress on Services*. Beijing, China, Oct. 2020. URL: https://inria.hal.science/hal-02983557.