C›ONSTRUCTOR
UNIVERSITY

Local-first Collaboration on Relational Data

by

Veronika Sirotkina

Bachelor Thesis in Computer Science

Submission: May 4, 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

Nowadays many apps need an internet connection to work properly, especially those that sync data between devices and let people collaborate in real-time. Without internet these apps often become unusable. Local-first software aims to change that by letting users edit shared documents offline and syncing changes when they are back online.

This paper explores the ways of enabling local-first real-time collaboration over abstract relational data in the context of GanttProject, an open-source tool for creating Gantt charts.

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 [14].

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 [3].

Local-first software [10] 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 [6, 5] 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 [4]. 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

2.1 Operational Transformation

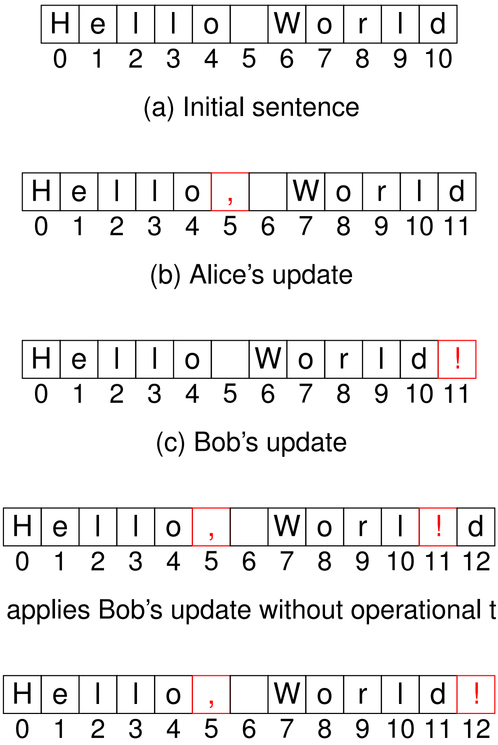
Operational Transformation [14] or OT is a technology for managing concurrent updates, particularly in collaboration software. OT algorithms assume that there are multiple clients. All clients have a replica of the same document and they can modify the document with a predetermined set of operations independently from each other.

When a client receives an update from another client, he doesn't apply the received 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

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)`.



(a) Initial sentence

(b) Alice's update

(c) Bob's update

(d) Alice applies Bob's update without operational transform

(e) Alice applies Bob's update with operational transform

Figure 1: Demonstrating Operational Transformation on plain text

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 [7, 8, 16] 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 [10] 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. For example, take an online LaTeX editor called Overleaf. In Overleaf if the connection is lost, the page waits for a few seconds before it tries to connect again. In those few seconds the user is still allowed to modify the text fields, but when connection is re-established, the user is met with an immediate error, notifying them that the project went out of sync. Overleaf makes no effort to integrate the changes made by the user while they were offline. This leads to a very frustrating experience where local changes are either discarded or the user has to manually copy them from a special pop-up.

  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 [3]. 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 [13]. 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. You can see an example implemetation of such counter on Figure 2

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, $C(merge(x_i, x_j)) = C(x_i) \cup C(x_j)$.

The example implementation of a Counter CRDT on Figure 2 is state-based.

2.3.2 Operation-based CRDT

Operation-based CRDTs are not required to have a merge function, but their updates must consist of two phases. The first phase called `atSource` is executed completely at a local replica and produces no side-effects. The second phase called `downstream` is executed locally immedately after the `atSource` phase. After that the `downstream` update is propagated to other replicas and is executed asynchronously by them, changing the downstream state.

```kotlin
import java.lang.Integer.max

const val MAX_REPLICAS = 3

class StateCounter(
    // One of {0, 1, ..., MAX_REPLICAS - 1}
    // The problem of assigning a unique id
    // is not considered in the context of this example
    private val id: Int = 0
) {
    // Initial state
    private val counter = MutableList(MAX_REPLICAS) { 0 }

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

    // Query
    fun value() = counter.sum()

    // Merge function
    fun merge(other: StateCounter) {
        for (i in 0 until MAX_REPLICAS) {
            counter[i] = max(counter[i], other.counter[i])
        }
    }
}
```

Figure 2: Example implementation of a state-based Counter CRDT in Kotlin

```kotlin
import kotlin.random.Random

typealias OperationId = Long

class OperationCounter {

    // Initial state
    private val increments = mutableSetOf<OperationId>()

    // Update operation
    fun increment() {
        val operationId = incrementAtSource()
        incrementDownstream(operationId)
    }

    // atSource phase of the increment operation
    private fun incrementAtSource(): OperationId {
        return Random.nextLong()
    }

    // downstream phase of the increment operation
    fun incrementDownstream(operationId: OperationId) {
        increments.add(operationId)
    }

    // Query
    fun value() = increments.size
}
```

Figure 3: Example implementation of a state-based Counter CRDT in Kotlin

This specification can be interpreted in the following way: the `atSource` phase generates an update that follows the internal protocol of the respective CRDT, while the `downstream` phase tries to apply the update to the object.

Figure 3 demonstrates an example implementation of an operation-based Counter CRDT.

Causal history can also be defined for operation-based CRDTs.

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

- Initially, $C(x_i) = \emptyset$.
- After executing the downstream phase of operation f at replica x_i, $C(f(x_i)) = C(x_i) \cup \{f\}$.

TODO: ELABORATE ON THE USAGE OF DEFINITIONS AND DEFINE CONVERGENCE OR DROP THEM COMPLETELY

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.

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 [11].

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 an example using Automerge's Rust API demonstrating Conflict objects on Figure 4

2.3.4 CRDT for Relational Data

Some work on designing CRDT for relational data have been done before [17]. 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.

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

fn main() {
    // Create an automerge document and its replica
    let mut first_doc = automerge::Automerge::new();
    let mut second_doc = first_doc.fork();

    // Set property "key" to "first value" in the original document
    first_doc.transact::<_, _, automerge::AutomergeError>(|doc| {
        Ok(doc.put(automerge::ROOT, "key", "first value").unwrap())
    }).unwrap();
    // Set property "key" to "second value" in the replica
    second_doc.transact::<_, _, automerge::AutomergeError>(|doc| {
        Ok(doc.put(automerge::ROOT, "key", "second value").unwrap())
    }).unwrap();

    // Replicas merge each other's changes
    first_doc.merge(&mut second_doc).unwrap();
    second_doc.merge(&mut first_doc).unwrap();

    // Get the conflict object
    let conflict1: Vec<_> = first_doc
        .get_all(automerge::ROOT, "key").unwrap();
    // Get the resolved conflict value
    let value1 = first_doc.
        get(automerge::ROOT, "key").unwrap().unwrap().0;

    // Get the replica's conflict object
    let conflict2: Vec<_> = second_doc
        .get_all(automerge::ROOT, "key").unwrap();
    // Get the resolved conflict value
    let value2 = second_doc
        .get(automerge::ROOT, "key").unwrap().unwrap().0;

    println!("{:?}", conflict1);
    // [(Scalar(Str("first value")), Id(1, ActorID("aaa"), 0)),
↪   (Scalar(Str("second value")), Id(1, ActorID("bbb"), 1))]
    println!("{:?}", conflict2);
    // [(Scalar(Str("first value")), Id(1, ActorID("aaa"), 1)),
↪   (Scalar(Str("second value")), Id(1, ActorID("bbb"), 0))]
    println!("{:?}", value1); // Scalar(Str("second value"))
    println!("{:?}", value2); // Scalar(Str("second value"))
}
```

Figure 4: Conflict objects in Automerge/Rust

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. A paper on CRDT [13] suggests that this problem could be solved by introducing quotas for certain operation. For example, in case of a counter supports both increments and decrements, but mustn't go below zero, let's impose the following restriction: a client can't issue more decrements than the number of increments he personally issued. If a device wants to decrement more, it can contact other clients to redistribute the allowed decrements. Still, this restriction is overly strong. The other solution to maintain the global invariant is to synchronize, but that defeats the purpose of CRDT.

3 Approaches

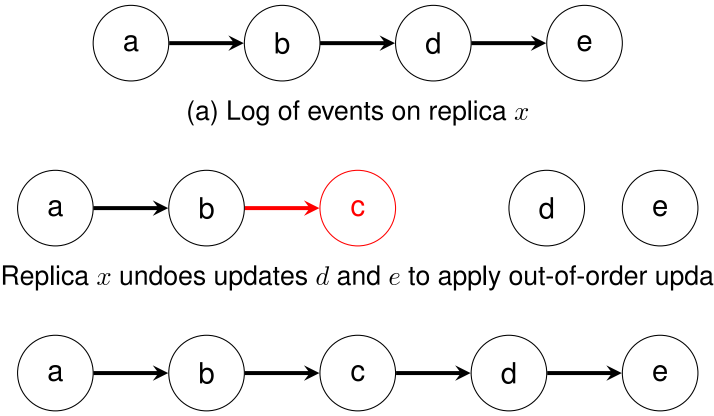
In this section I will describe approaches considered during the course of my work.

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.



(a) Log of events on replica x

(b) Replica x undoes updates d and e to apply out-of-order update c

(c) Replica x reapplies updates d and e on top of c

Figure 5: Demonstration of time warp progression

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. See Figure 5 for a demonstration of time warping.

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. receive 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. receive 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 sends 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. TODO: MAYBE UNCLEAR, ADD PICTURE?

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 most relational databases, including Postgres. To allow time warp on a database, one would have to be able to generate undo operations for all possible transactions. One notable challenge is recovering rows that where deleted as a side-effect of removing a foreign key they were referencing. Another challenge comes from the fact that transactions usually can't be organized into a strictly ordered sequence. In Postgres, multiple transaction isolation levels are supported, and only the strongest level, called SERIALIZABLE, has this property. See Serializable Isolation Level documentation [12] for more details.

3.3 Transaction replay approach

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 [12]. 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 row
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 row
-- another transaction commits deleting a row
SELECT * FROM data; -- 1 row
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.

3.4 Data Branching

Data branching in databases refers to the practice of creating divergent versions of data within a database, often for the purpose of testing, experimentation, or parallel development. This concept is similar to branching in version control systems like Git, but applied to data within a database rather than code in a repository.

Data branching could solve some problems posed by the Time Warp and Transaction replay approaches. In case of Time Warp, there is no need to undo operations if there exists a branch that contains the required version of the project. In case of Transaction replay, there would be no need to store the state of the project after each transaction.

One example of data branching for Postgres is Neon [2]. Unfortunately, the free version of Neon only allows to create 10 branches, which is not enough for the purposes of GanttProject.

TODO: WRITE MORE?

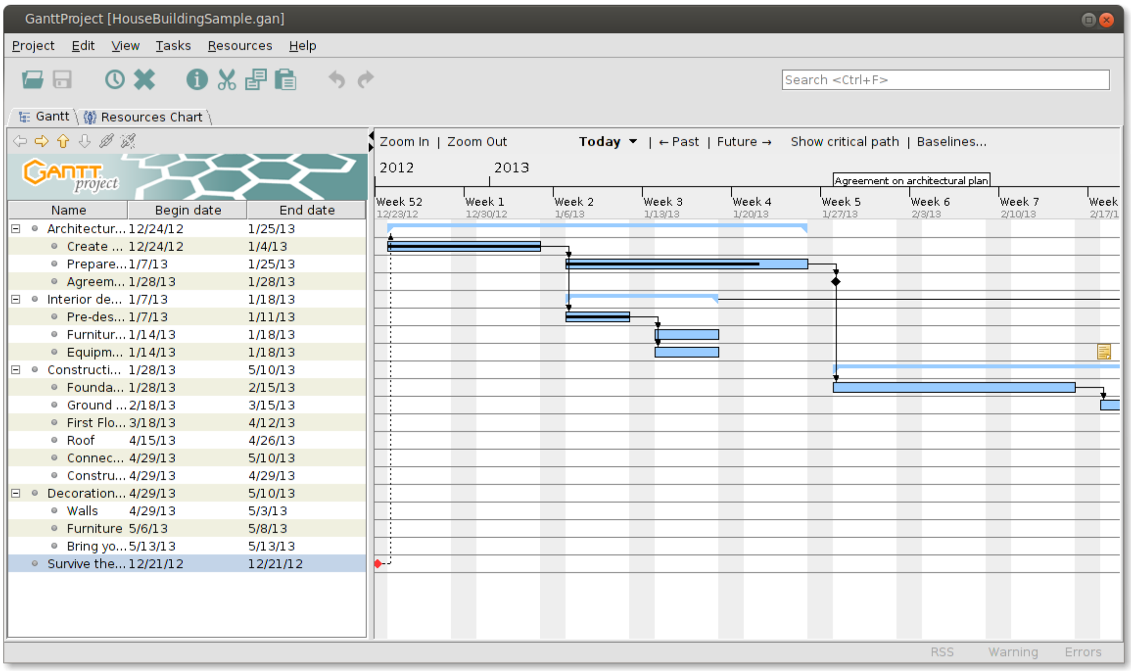
4 GanttProject

GanttProject is an open-source desktop application for creating Gantt charts. A Gantt chart is a type of bar chart that illustrates a project schedule. It's a visual representation of the tasks and activities that need to be completed, along with their corresponding start and end dates. Gantt charts often include dependencies between tasks, showing which tasks must be completed before others can start. These dependencies are typically represented by arrows or lines connecting the bars. Some Gantt charts also include information about resource allocation, showing which resources (such as personnel or equipment) are assigned to each task.

GanttProject is written in Java and Kotlin and uses Postgres for the database. It supports saving projects in the cloud, but they can't be edited simultaneously by multiple people. GanttProject interface is shown on Figure 6

4.1 Colloboque

To enable real-time collaboration in GanttProject, a module called Colloboque is being developed. It is written in Kotlin. Before I started working on Colloboque, it worked as



Name	Begin date	End date
Architectur...	12/24/12	1/25/13
Create ...	12/24/12	1/4/13
Prepare...	1/7/13	1/25/13
Agreem...	1/28/13	1/28/13
Interior de...	1/7/13	1/18/13
Pre-des...	1/7/13	1/11/13
Furnitur...	1/14/13	1/18/13
Equipm...	1/14/13	1/18/13
Constructi...	1/28/13	5/10/13
Founda...	1/28/13	2/15/13
Ground ...	2/18/13	3/15/13
First Flo...	3/18/13	4/12/13
Roof	4/15/13	4/26/13
Connec...	4/29/13	5/10/13
Constru...	4/29/13	4/29/13
Decoration...	4/29/13	5/10/13
Walls	4/29/13	5/3/13
Furniture	5/6/13	5/8/13
Bring yo...	5/13/13	5/13/13
Survive the...	12/21/12	12/21/12

Figure 6: GanttProject interface

follows.

Clients connect to a centralized Colloboque server and the server responds with an XML file describing the current version of the project. It was supposed to be the last version of the project, but the stub implementation responded with the same default project every time. The client received and loaded the project XML sent by the server. After that every change made by the user is wrapped into an update message and sent to the server. The client waits for the server to respond with a commit message. Until a response from the server is received, new changes are not allowed.

The server receives messages from clients. Each message contains information about a transaction that the client wants to perform. The server assigns a transaction id to each transaction, and the client's message includes the transaction id that the client last observed. If the server's last transaction id and the base transaction id in the update message do not match, the server informs the client about the error.

If the transaction ids match, the server executes the transaction, generates a commit message and propagates it to all connected clients.

This implementation had a few problems:

1. Imagine a simple situation. A new project is created, let's denote the initial state of the project as p. Then one client connects and makes a change, producing a second version of the project - let's denote the second version p'. After that a second client connects to the same project. While it is expected that the server would share the latest version of the project p' with the new client, this wasn't the case. In reality, the server would always respond to all newly connected clients with the same default project template p.

2. The transactions performed by the server were not logged. This means that if a

```
CREATE TABLE Task (
    uid                  VARCHAR            NOT NULL,
    num                  INTEGER            NOT NULL,
    name                 VARCHAR            NOT NULL,
    color                VARCHAR                NULL,
    shape                VARCHAR                NULL,
    is_milestone         BOOLEAN            NOT NULL DEFAULT false,
    is_project_task      BOOLEAN            NOT NULL DEFAULT false,
    start_date           DATE               NOT NULL,
    duration             INTEGER            NOT NULL,
    completion           INTEGER                NULL,
    earliest_start_date  DATE                   NULL,
    priority             VARCHAR            NOT NULL DEFAULT '1',
    web_link             VARCHAR                NULL,
    cost_manual_value    NUMERIC(1000, 2)       NULL,
    is_cost_calculated   BOOLEAN                NULL,
    notes                VARCHAR                NULL,

    PRIMARY KEY (uid)
);
```

Figure 7: Task table definition on a GanttProject client

client was disconnected during some update, then this update is completely lost for this client. The server doesn't have the ability to generate an XML file describing the project from the database state, so recovery was not possible in this situation.

3. When a client performs and update, he has to wait for confirmation from the server. During that time he can't introduce more updates. This makes collaboration in GanttProject not local-first. While it was never intended to be local-first in the first place, it is a nice feature that can be introduced.

4.2 Data model

Due to the way relational databases work, conficts may arise in places where, intuitively, they shouldn't. For example, if one client tries to change task duration, while another tries to change task description, they will run into a conflict. This is because in the database all information about a task is stored in a single row. When clients try to concurrently change task duration and task description, they are essentially trying to change the same row. This will lead to an error if the two transactions happen to run in parallel on the server.

To avoid that, the data model on the Colloboque server differs from the one on the client. The client's `Task` table can be found on Figure 7. On the server this table is separated into multiple tables, each table containing `uid` of the task and some of the attributes of the original `Task` table, grouped either by logic or attribute type. A view object joins all the tables and provides an interface that allows to query all the different tables as if they where a single `Task` table. Additionally, triggers are specified to handle updates and insertions called on the view, so that they are performed on the underlying tables. The server's `Task` view definition can be found on Figure 8

```
-- Task start date and duration shall be changed as a whole
CREATE TABLE TaskDates (
    uid                 VARCHAR(128) PRIMARY KEY REFERENCES TaskName,
    start_date          DATE         NOT NULL,
    duration_days       INT          NOT NULL DEFAULT 1,
    earliest_start_date DATE
);

-- Integer valued properties
CREATE TABLE TaskIntProperties (
    uid                 VARCHAR(128) REFERENCES TaskName,
    prop_name           TaskIntPropertyName,
    prop_value          INT,
    PRIMARY KEY (uid, prop_name)
);

----------------------------------------
-- Other component tables are omitted --
----------------------------------------

-- Updatable view which collects all task properties in a single row.
-- Inserts and updates are processed with INSTEAD OF triggers.
CREATE VIEW Task AS
SELECT
    TaskName.uid,
    num,
    name,
    start_date,
    duration_days AS duration,
    earliest_start_date,
    is_cost_calculated,
    cost_manual_value,
    is_milestone,
    is_project_task,
    MAX(TIP.prop_value) FILTER (WHERE TIP.prop_name = 'completion' AS completion,
    MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'priority')  AS priority,
    MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'color')     AS color,
    MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'shape')     AS shape,
    MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'web_link')  AS web_link,
    MAX(TTP.prop_value) FILTER (WHERE TTP.prop_name = 'notes')     AS notes
FROM TaskName
        JOIN      TaskDates USING (uid)
        LEFT JOIN TaskIntProperties TIP USING (uid)
        LEFT JOIN TaskTextProperties TTP USING (uid)
        LEFT JOIN TaskCostProperties TCP USING (uid)
        LEFT JOIN TaskClassProperties TCLP USING (uid)
GROUP BY TaskName.uid, TaskDates.uid, TCP.uid, TCLP.uid;
```

Figure 8: Task view definition on a GanttProject Colloboque server

Specification 1 Update object

```
    id: Id                    // Unique identifier of this update
    baseId: Id                // Id of the previous update
    author: String            // Id of the client who initiated the update
    transaction: Transaction // Transaction to apply to the database
    status: UpdateStatus      // One of { COMMIT, REJECT, LOCAL }
                              // COMMIT: committed by the server
                              // REJECT: rejected by the server
                              // LOCAL:  not processed by the server yet
```

5 Synchronization Protocol

5.1 Outline

The client can generate local updates and accept updates from the server. Local and server updates are processed synchronously by the client. Local updates are communicated to the server after the client applies them. The server processes all client updates synchronously and establishes a total order of the events. The client has to follow the total order of the events established by the server. The client can deviate from the established order because it applies local updates without getting confirmation from the server. Upon discovering errors in the order of applied events, the client enters synchronization phase. In synchronization phase the client stops accepting updates. It takes all local updates that are not confirmed by the server yet, and sends them to the server. The server uses Transaction Replay approach (TODO: CITE THE CORRESPONDING SECTION) to check whether applying the updates produces conflicts. If it doesn't produce conflicts, the server applies the updates. Finally, the server responds to the client with a list of updates that the client must apply to get the latest state of the project and an indication whether the updates that the client sent were applied successfully.

5.2 Database model

Consider that all database transactions are deterministic and run with isolation level SERIALIZEABLE. Then the database state can be accurately described by a linear history of transactions. In more formal terms, project database is *simulated* by a linear history of transactions. (TODO: DEFINE FORMALLY?) The protocol specification uses an append-only log of updates to represent the history of transactions and describe the state of the database.

5.3 Server

Logs are composed of `Update` objects. `Update` object specification can be found in Specification 1. `Log` object specification can be found in Specification 2. The server's state and API can be found in Specifications 3 and 4

When the server receives a `sendUpdate` call, it first compares `update.baseId` against `log.lastUpdateId()` (the last update id in the log). If the ids match, the server commits the update and notifies all connected clients about the update. If the ids don't match, the server notifies the author of the update about their update being rejected.

When the server receives a `synchronize` call, it checks whether the updates produce

Specification 2 Log object

```
State:
    updates: List<Update>

Interface:
    fun append(update: Update):
        updates.add(update)

    fun lastUpdateId():
        return updates.last().id
```

Specification 3 Server state

```
log: Log
connectedClients: List<ClientConnection>
```

conflicts. The exact logic of conflict checking is omitted in this protocol specification to avoid increasing complexity, but in the implementation the check for conflicts is performed using Transaction Replay. If there are no conflicts, the server changes the first update's `baseId` to match the current log and applies the updates. If there are conflicts, the updates are discarded. Finally, the server computes a list of updates that the client needs to apply locally in order to get the latest state.

5.4 Client

To provide seamless integration of local updates and updates from the server, the client stores two logs. The first log is the client's personal log, to which local updates can be written without confirmation from the server. This is also what's displayed to the user. The second log strictly follows the server's log, and its purpose is to make synchronization process more efficient. The client also stores a list of updates that were applied to the personal log, but haven't received confirmation from the server.

When a client connects to the server for the first time, it fetches the latest log using the server's `fetchLog` API call. After that the server will send the client notifications about updates in the order that the server processes them. Each notification contains a single `Update` object. The client can also generate local updates. To avoid complexity of distributed applications, local updates and server updates are merged in a single queue, so the client can process all updates synchronously.

The client processes each update based on its `status` field. There are three update handlers in total: `processLocalUpdate`, `processServerUpdateCommitted` and `processServerUpdateRejected`. Definitions of these handlers can be found in Specification 6

Local update handler

The `processLocalUpdate` handler manages updates generated locally by the client. It appends the update to the client's log (`clientLog`) and adds it to the list of unconfirmed updates (`unconfirmedUpdates`). It then notifies the server of the update using the `sendUpdate` API call.

Specification 4 Server API

```
    sendUpdate(update: Update)

    synchronize(unconfirmedUpdates: List<Update>, lastConfirmedId: Id):
        returns (updates: List<Update>, updatesApplied: Boolean)

    fetchLog:
        returns (log: Log)
```

Specification 5 Client state

```
    clientId: String                  // Unique id of this client.
    clientLog: Log                    // Client's personal log.
    serverLog: Log                    // Replica of the server's log.
    unconfirmedUpdates: List<Update>  // Updates that were applied to
                                      // the personal log, but not
                                      // confirmed by the server.
    server: ServerConnection
```

Commit update handler

The `processServerUpdateCommitted` handler is used for updates that were sent as commit notifications by the server. The update is either a confirmation for this client's local update or a new update from another client, processed by the server. We can look at this update from two perspectives: from the perspective of the `clientLog` and from the perspective of the `serverLog`.

From the perspective of the `serverLog`, the update can be either one of three cases:

1. The update is an older update that has already been confirmed. This can happen because the client may get update confirmations before the notification arrives by using `fetchLog` and `synchronize` calls. No actions are required in this case.

2. The update's `baseId` is the same as the log's last update id. The update can be appended directly to the log.

3. The situation doesn't match any of the other cases. This implies that some notifications from the server have been lost. It is only possible in case of this client's or a server's failure and is omitted here.

An illustration of these cases is displayed on Figure 9

From the perspective of the `clientLog`, the update can be either one of five cases:

1. The update is an older update that has already been confirmed. This can happen because the client may get update confirmations before the notification arrives by using `fetchLog` and `synchronize` calls. No actions are required in this case.

2. The update confirms the oldest unconfirmed update. In this case it should be removed from the `unconfirmedUpdates` list and no further actions are required.

3. The update's `baseId` is the same as the log's last update id and the list of unconfirmed updates is empty. The update can be appended directly to the client's log.

4. The update's `baseId` is the same as the oldest unconfirmed update's `baseId`, but

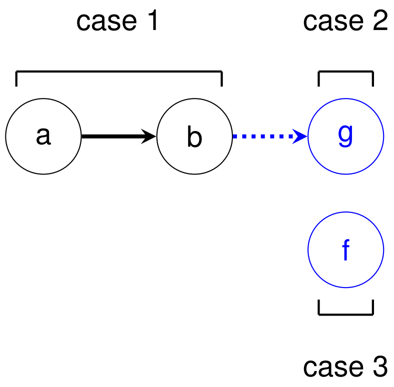


Figure 9: Visualization of cases in relation to `serverLog`.

Black nodes represent the current log. Blue nodes represent updates that don't match any update in the log. An arrow from `x` to `y` shows that `x.baseId == y.id`.

the updates are different. This implies that the client has diverged from the server and must synchronize.

5. The situation doesn't match any of the other cases. This implies that some notifications from the server have been lost. It is only possible in case of this client's or a server's failure and is omitted here. TODO?

An illustration of these cases is displayed on Figure 10

Reject update handler

The `processServerUpdateRejected` handler is called when the server rejects an update previously sent by the client. It ensures that the client has already received the conflicting update notification and completed synchronization, so no further action is necessary.

5.5 Synchronization phase

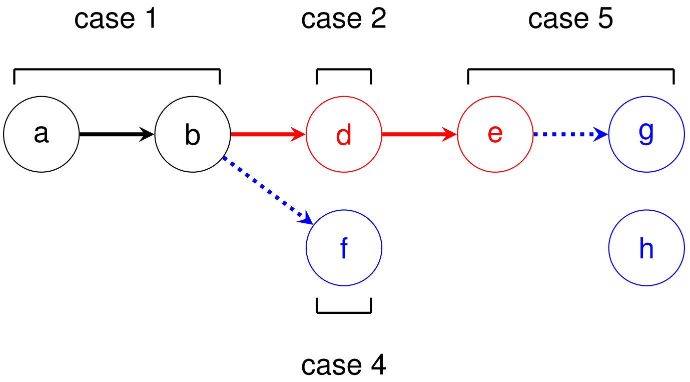
The synchronization phase consists of the following steps:

1. The client sends to the server all unconfirmed updates and an id of the last server-confirmed update that the client observed. After that the client waits for the server's response.

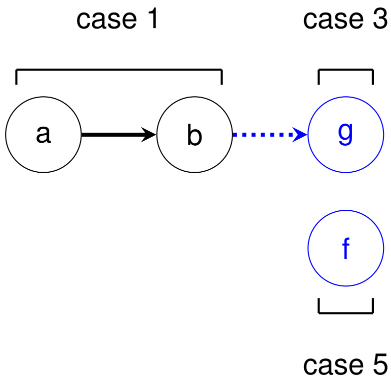
2. The server receives `unconfirmedUpdates` from the client and checks whether applying them would produce conflicts. Transaction Replay is used to do that, but I will omit implementation details. If there are no conflicts, the updates are applied, otherwise they are discarded.

3. The server uses `lastConfirmedId` and the current log data to compute a list of updates that the client needs to apply to get the latest state of the project. The server sends this list to the client together with an indication whether `unconfirmedUpdates` where applied.

4. The client applies received updates to `serverLog`, copies `serverLog` into `clientLog`



(a) Unconfirmed updates are present.



(b) Unconfirmed updates are not present.

Figure 10: Visualization of cases in relation to `clientLog`.

Black and red nodes show the client's log. Red nodes represent unconfirmed updates. Blue nodes represent updates that don't match any update in the log. An arrow from `x` to `y` shows that `x.baseId == y.id`.

and empties `unconfirmedUpdates`. Alternatively, in case the updates weren't applied, the user can be prompted to resolve conflicts manually.

6 Implementation

A prototype of the protocol described in Section 5 was implemented[15]. In the implementation WebSockets are used to communicate updates from clients to the server and from the server to clients, and HTTP is used for synchronization and fetching of the log. This architecture mimicks the architecture of GanttProjects's Colloboque to ease integration with the main codebase.

Additionally, some changes were made to Colloboque to facilitate new protocol implementation.

jOOQ

Firstly, the use of jOOQ library [9] was integrated into Colloboque. jOOQ can generate Java classes based on the database schema, describing tables, columns, and so on. These generated classes provide type-safe access to the database objects, eliminating the need for manual SQL string concatenation and reducing the risk of SQL injection attacks. jOOQ also abstracts away the differences between SQL dialects. This allows to write database-agnostic code that can be easily ported.

Restoring the latest version

Secondly, persistent loggigng of executed transactions and the project's state was implemented. This would allow the server to use Transaction Replay to check for conflicts. When conflicting updates arrive, the server can use `baseId` to load the state of the project at `baseId` from the project state logs, as well as a list of concurrent updates from transaction logs.

6.1 Transaction replay

Thirdly, a function to run transaction replay was implemented.

References

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

[2] *Data branching for Postgres by Neon*. https://neon.tech/branching.

[3] 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/.

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

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

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

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

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

[9] *jOOQ library*. https://www.jooq.org/.

[10] 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.

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

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

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

[14] 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.

[15] *Synchronization protocol prototype*. https://github.com/ozzush/local-first-collab-protocol-prototype.git.

[16] David Wang, Alex Mah, and Soren Lassen. *Google Wave Operational Transformation*. https://svn.apache.org/repos/asf/incubator/wave/whitepapers/operational-transform/operational-transform.html. July 2010.

[17] 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.

Specification 6 Update handlers

```
// LOCAL
processLocalUpdate(update: Update):
    // Implementation ensures that
    // update.baseId == lastClientUpdateId
    clientLog.append(update)
    unconfirmedUpdates.add(update)
    server.sendUpdate(update)

// COMMIT
processServerUpdateCommitted(update: Update):
    if (update.baseId == serverLog.lastUpdateId()) {
        serverLog.append(update)
        if (update.baseId == clientLog.lastUpdateId()) {
            clientLog.append(update)
        } else if (update == unconfirmedUpdates.first()) {
            unconfirmedUpdates.removeFirst()
        } else if (update.author == clientId) {
            // Do nothing.
            // Confirmation for this update was already received
            // through a `fetchLog`' or `synchronize`' call
        } else {
            synchronizationPhase()
        }
    } else {
        // Either some updates were lost due to this client's failure,
        // in which case we should synchronize,
        // or the update was already applied by this client,
        // in which case the update should be ignored.
        //
        // The latter can happen because the client receives
        // updates ahead of commit notifications in response to
        // `fetchLog` and `synchronize` calls.
    }

// REJECT
processServerUpdateRejected(update: Update):
    // Do nothing.
    // The client processes updates in the same order as
    // the server issues them.
    // Rejection is only possible if another update
    // with the same baseId was processed by the server earlier.
    // When this rejection  arrives, the client has already received
    // the conflicting update notification and finished
    // synchronization.
```