Ruprecht Karls University Heidelberg
Institute of Computer Science
Database Systems Research Group

Bachelor Thesis

Enabling Offline Usage and Synchronization in Mobile and Collaborative Apps

Name: Mirko Kiefer

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

Date of Submission: April 19, 2013

Abstract

Zusammenfassung

Contents

1 Introduction

1.1 Motivation

Applications that allow users to collaborate on data on a central server are in widespread use. Popular examples are document authoring tools like Google Docs, project collaboration apps like Basecamp or Trello or even large scale collaboration projects like Wikipedia.

1.1.1 The Thin-Client, Fat-Server Model

The traditional architecture of these applications follows a client-server model where the server hosts the entire application logic and persistence. Users access the application through a thin client, most commonly a web browser. The browser only has display user interfaces that are pre-rendered by the server.
This model works well when using desktop computers with a realiable, high-speed connection to the server.

1.1.2 The Fat-Client, Fat-Server Model

Rising expectations in usability drove developers to increasingly move application logic to the client. Initially this has only been the logic required to render the user interfaces. The server still hosted most of the application logic to pre-compute all relevant data for the client.
Moving the interface rendering to the client reduces the amount of data that has to be transferred and makes the application behave more responsive.

1.1.3 The Fat-Client, Thin-Server Model

The widespread adoption of mobile devices forces developers to re-think their architecture again. Users can now carry their devices with them and expect their applications

to work outside their home or office network. Applications therefore have to work with limited mobile internet access or often no access at all.
The only way to support this is by moving more of the application logic to the client and by replicating data for offline use. The clients are now not only responsible for rendering interfaces but also implement most of the application logic themselves.
The new architecture comes at a high price - the additional client logic and persistence adds a lot of complexity. While in the thin client-fat server model developers only had to maintain a single technology set on the server, they now face different technologies on each platform they aim to support with a fat client.
The ability to use the application offline requires an entire new layer of application logic to manage the propagation and merging of changes and to resolve conflicts.
The only responsiblity of the server in this model is the propagation of data between clients.

1.1.4 The Client as a Server Model

Most users today carry a notebook, a smartphone and maybe even a table computer with them. They often want to work with the same data on different devices. Apps need to support workflows like adding some items to a Todo-Manager on a notebook and subsequently reviewing them on a smartphone. Today's applications only achieve this through data synchronization between the devices and a central server. If the user is mobile and does not have reliable internet connection he is stuck with outdated data on his smartphone.
This problem can only be resolved by supporting the direct synchronization between devices. The clients can now bascially act as servers themselves and manage propagation of data to other clients.
The actual server does not have to disappear in this model. But just as the clients he is just another node on the network. The difference is that he is continuously connected to the internet and can therefore play a useful role as a fallback.
Note that this only describes the extreme case - in most real-world applications we will see a hybrid-architecture where clients can synchronize most data directly but the server still manages security or enforces other constraints.
Building such a distributed data synchronization engine is very complex and beyond the reach of a small team of app developers. It is also way beyond the scope of this thesis. As described in the next section we will focus on a set of use cases and questions. - add Things story on how hard it is

1.2 Goals of the thesis

This thesis aims to develop patterns and tools to make the development of offline capable, collaborative apps more productive.

The guiding questions are:

- How can we enable the operation of a collaborative app with unreliable network connection?
- How can we efficiently synchronize changed data directly between unreliably connected devices?
- How can we abstract the synchronization logic to be as unintrusive as possible to an application?

A collaborative app that has to function with unreliable network connection implies that we can not rely on the traditional thin client model. We have to think about ways to make both data and logic available offline.

Being able to synchronize data directly between devices forces us to develop a distributed architecture.

Efficient synchronization means that we aim to minimize the amount of redundant data sent between devices. We have to figure out ways to identify changes in the data.

Combined with the requirement to be unintrusive we exclude solutions that require the application to explicitly track changes in the code. The identification of data changes should be decoupled from the main application logic. This ensures that an upgrade of traditional applications requires minimal effort.

We will refine this set of requirements by breaking down common use cases and evaluating existing solutions that support offline-capable applications.

1.3 Structure of the thesis

Here you describe the structure of the thesis. For example:

In Kapitel 2 werden grundlegende Methoden für diese Arbeit vorgestellt.

2 Background

Here you discuss some basics for your work and outline existing research in the area of your thesis by citing research papers like [1] by Lindholm and [2, 3] Candia.

2.1 Causality Preservation

2.2 Optimistic Synchronization

2.3 Vector Clocks

2.4 Operational Transformation

2.5 Commutative Replicated Data Types

2.6 State vs. Edit-Based Syncing

2.7 Three-Way Merging

- 3DM tool (Lindholm)

2.8 Most Recent Common Ancestor

3 Designing a Synchronization Framework

3.1 Application Scenarios

We describe common synchronization scenarios based on popular mobile applications.

3.1.1 A Collaborative Task Manager

The seemingly simple scenario of a Task Manager turns out to be very complex to implement if enhanced with collaborative features and offline availability.

Let us first look at some user stories and then try to define a suitable data model for such an application.

User Story 1: Creating Projects

- A User can create Projects in order to coordinate Tasks.
- A User can invite other Users as Project Members to a Project.

Examples for Projects created by User Rita would be:

Project Name	Members
Marketing Material	Rita, Tom, Allen
Product Roadmap	Rita, Allen
Sales Review	Rita, Lisa

User Story 2: Creating and Editing Tasks

- Project Members can add Tasks to a Project in order to manage responsibilities.
- A Task can have a due date and responsible project member assigned.
- A Task can be edited by Project Members and marked as done.

- A Task can be moved in the list of Tasks.

An example list of Tasks could be:

Project “Marketing Material”

Task	Due Date	Assignee	Done
Create event poster	2013-08-12	Rita	No
Write blog entry on event	2013-07-20	Tom	Yes

User Story 3: Commenting on Tasks

- Project Members can add Comments to Tasks

Examples would be:

Task “Create event poster” in Project “Marketing Material"

Member	Date	Comment
Rita	2013-07-20	Allen, I need you to create some graphics.
Allen	2014-07-20	Ok, lets go through it tomorrow morning!

User Story 3: User Workflows

- In order to be productive a user needs to access all Tasks from any device.
- A user should be able to Edit and Create Projects and Tasks when disconnected from any network.
- The data should be kept as current as possible even if a user’s device does not have reliable internet access.

An example workflow that should be supported:

- Rita works at the desktop computer in her office with high-speed internet access. She creates Project A and invites Allen.
- Allen works from home on his notebook with high-speed internet access. He reviews the Project and creates task A1.
- Rita is already on her way home but has mobile internet access on her smartphone. She receives the added task A1 and edits its title.

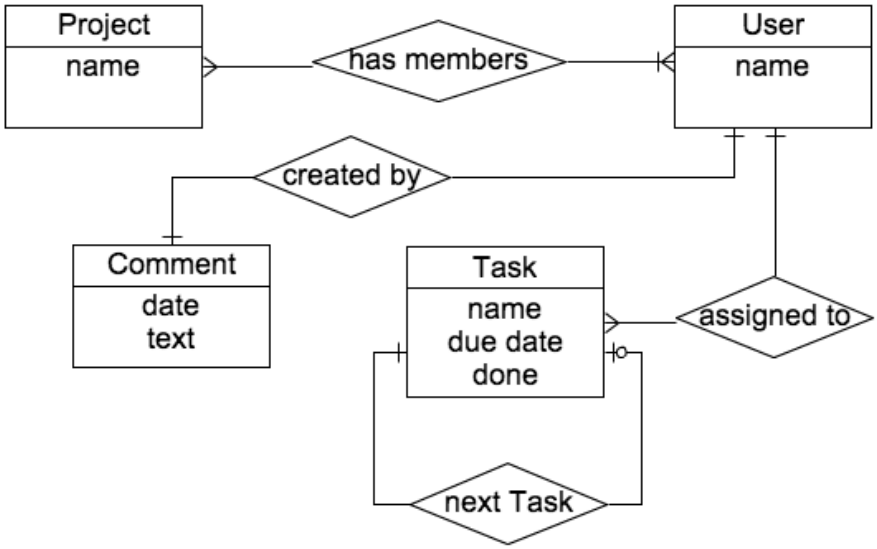

Figure 3.1: A collaborative Task Manager's data model

- Rita is still on the train but decides to continue working on her notebook. Her notebook does not have internet access but she can establish a direct connection to her smartphone via Wifi. The reception on her smartphone has dropped in the meanwhile. She receives the latest updates from her smartphone and adds a comment to task A1.
- Allen who is still at home can not receive Rita's comment as she is still on the train. In the meanwhile he creates a Task A2 in Project A.
- Rite gets home where she has internet access with her notebook. She receives Allen's created Task A2.
- Allen, who is still at his notebook, receives Rita's comment as soon as she connects to internet at home.

Data Model

Based on the user stories we can derive a plausible data model for the application. We can map it to a straight-forward entity-relationship model as shown in figure 3.1.
The only complication is the requirement of Tasks per Project being ordered. We model this as a linked list by having a "Next Task" relationship.

3.1.2 A File Sharing App

Dropbox synchronizes a file system - it is therefore a good example for syncing of hierarchical data.

The data model is simple:

- Tree Item (name)
- Tree extends Tree Item (has children of type Tree Items)
- Data extends Tree Item (data)

The list of child Tree Items can either be ordered or unordered. While Dropbox does not sync the order of files there are scenarios where this is required.

Syncing trees can trigger conflicts if sub trees have been modified concurrently.

3.1.3 (Text Synchronization)

Collaborative document editors like Google Docs need to synchronize text that is concurrently edited.
Google Docs currently does not support offline editing.

Syncing text is equal to the problem of syncing an ordered list and can trigger conflicts.

3.2 Requirements

From the application scenarios we can derive a set of requirements for a synchronization solution.

Phases of synchronization: - Update Detection - Update Propagation - Reconciliation

3.2.1 Flexible Data Model Support

A synchronization engine that is useful for a broad range of applications has to be able to deal with different data models. There is no magic algorithm that produces a perfect solution for an existing application. Synchronization can happen with increasing levels of sophistication depending on the amount of understanding of an application's data.
A "dumb" solution would have no understanding of an app's data model at all - it simply sees the entire application data as one binary chunk.
A more clever solution would maybe have an understanding of entities like Projects, Tasks or Comments and would see the entity instances as binary data.
It could get even finer grained and break up each entity instance into attributes which

it recognizes as different pieces of data.
We see that *synchronization granularity* is one key aspect when evaluating solutions. The smallest pieces of information a syncing engine can not break up further we call *atoms*. Atoms are usually aggregated into larger structures we call *objects*. A Task instance could be treated as an object which composes the title and due date attributes as atoms.
In order to be useful a synchronization engine does not need perfect understanding of the data to be synchronized. Popular applications like Dropbox can provide useful synchronization of files without having any semantic understanding of their content. For Dropbox each file is an atom - if a user adds a paragraph to a Word document, Dropbox only recognizes a change to the entire file. This means if two users concurrently modify the same document at different places, Dropbox has no way to merge the changes correctly and will trigger a conflict.
Version control systems like git are usually more sophisticated - git treats each line in a file as an atom and can therefore often successfully merge concurrent changes. Git still does not have any syntactic or even semantic understanding of the code that is written in the files it synchronizes. So if there are concurrent edits, git can not guarantee that merges are synctactically or semantically correct. Despite this git is useful enough to be used very successfully in large software projects.

We therefore need to evaluate carefully how much semantic understanding of an application's data a synchronization engine actually needs to be useful.

3.2.2 Optimistic Synchronization

As we have seen in the application scenario it is necessary that objects are editable on multiple devices even if they are not connected to a network. Edits should be allowed concurrently to not block users from doing their work. This implies that there can not be a central locking mechanism that controls when users can synchronize their data for offline usage. We therefore trade strict consistency for availability of the data.
Synchronization happens in an optimistic manner which means that we assume that temporarily inconsistent data will rarely lead to problems.

3.2.3 Eventual Consistency

The sequence of states an object goes through as its edited is called its *history*. The history forms a directed graph with each state except the intial state having at least one ancestor. The *current state* is the one that has no descendants. As edits can be made

on different devices concurrently there can be multiple *current states* at a time. If an object has multiple current states we refer to them as *branches*.
Our goal is to guarantee that after a finite number of synchronization events the object will eventually converge to the same state across all devices.

3.2.4 Causality Preservation and Conflicts

If an object diverges into multiple branches it will have to be reconciled during the synchronization process. When we receive states from a remote device we need to reason about how we can apply them to our own edit history.
The *happens-before* relationship defined by Lamport in [4] helps to reason about this problem in an intuitive way. A state a that *happened before* state b refers to the fact that the edits that led to b could have been affected by a. It is not necessarily related to the actual time of the edits that led to a and b as we can see in the following example:

Lets assume Rita and Allen work on the same object with their respective devices. The object has the initial state a.

- 9:00 AM: Rita makes an edit to the object which leads to state b.
- 9:30 AM: Allen synchronizes with Rita and edits which leads to state c.
- 10:00 AM: Rita is offline and can not synchronize. She edits the object at state b leading to state d.

As Allen has seen state b when making his edit, state b *happened before* state c.
Rita has not seen state c when making her edit. Although the time of her edit is after Allen's edit there is no *happened-before* relationship between state c and d.
On the next synchronization between Allen and Rita the system needs to identify this lack of causality as a *conflict*.
While this example is simple, the identification of conflicts among a large group of collaborators can be non-trivial.
Depending on the amount of understanding the synchronization engine has on the data there are strategies to resolve conflicts automatically. The engine should be designed in a way that conflict resolution strategies can be "plugged-in". If no automatic resolution is possible the application should be able to present the conflict to the user and let him manually resolve it.

3.2.5 Peer-to-Peer Synchronization

3.2.6 Integration with Existing Application Logic

Aspects to consider when evaluating strategies:

- How are updates detected?
- How are updates propagated? (Stream or Snapshot)
- How are updates merged/reconciled? (State or Edit-based)
- Level of structural awareness (Textual, Syntactic, Semantic/Structural)

3.3 Evaluating Existing Systems

Here we evaluate existing solutions based on the requirements.

3.3.1 git

- Data structure: filesystem/tree
- Merging: tree-based, three-way merge
- Propagation: snapshot-based
- Supports peer-to-peer

3.3.2 CouchDB

- Data structure: key-value
- Merging: tree-based
- Propagation: stream-based
- Supports peer-to-peer

3.3.3 (Backends-as-a-Service)

- parse.com - stackmob - deployd

Most of them simply expose a REST-API but leave all the conflict handling work to the app developer.
Are completely centralized.

3.4 Architecture of Synclib

Based on the requirements and the evaluation of existing systems we derive a unique architecture for a practical synchronization solution.

- **no timestamps**: state-based 3-way merging
- **no change tracing**: change tracing is not necessary - support diff computation on the fly
- **data agnostic**: leave diff and merge of the actual data to plugins
- **distributed**: syncing does not require a central server
- **be small**: only implement the functional parts of syncing - leave everything else to the application (transport, persistence)
- **sensitive defaults**: have defaults that *just work* but still support custom logic (e.g. for conflict resolution)

- cross-platform through web standards - solve server behaviour through native proxy - diff-merge-patch - most-recent-common-ancestor

3.5 Technologies used for Implementation

We describe implementation details like the technologies used, code structure and the testing framework to evaluate the system.

- everything web-based –> only way to be cross-platform - client-side persistence with HTML5 - note on alternatives (Lua, native)

3.6 Differencing and Merging of Data Models

- explain diff, merge and patch - implement diff, merge and patch logic for primitive data structures -> use them to recursively model complex data structures - ensure conflicts are made explicit

3.6.1 Sets

3.6.2 Ordered Lists

3.6.3 Ordered Sets

3.6.4 Dictionaries

used for object collections in data models

3.6.5 Ordered Dictionaries

most common for managing ordered object collections in data models can be modeled with dictionary and ordered set/list

3.6.6 Trees

- tree as an example for composite data model - efficient child tree pointers like in git

3.6.7 Composite Data Structures

- show how to represent complex data models as composite data structures

3.7 Storing and Commiting Changes

As syncing is state based we need to track the history of edits on each client.
Each client has his own replica of the database and commits data locally.
On every commit we create a commit object that links both to the new version of the data and the previous commit.

- use content-adressable store - only store changes and reference unchanged data through hashs –> like git - commit links to data and parent commit

3.8 Finding Common Commits

- Most Recent Common Ancestor algorithm used for finding common commit of clients
- described algorithm in background - implementation as separate module

3.9 Synchronization Protocol

If a client is connected to a server he will start the sync process on every commit. As synclib2's architecture is distributed a server could itself be a client who is connected to other servers.
To the latest commit on a database we refer to as the 'head'.

Synchronization follows the following protocol:

```
Client has committed to its local database.
Client pushs all commits since the last synced commit to Server.
Client asks Server for the common ancestor of client's head and the server's head
Client pushs all changed data since the common ancestor to Server.

if common ancestor == server head
  // there is no data to merge
  try fast-forward of server's head to client's head
  if failed (someone else updated server's head in the meantime) then start over
else
  Client asks Server for all commits + data since the common ancestor
  Client does a local merge and commits it to the local database
  start over
```

This protocol is able to minimize the amount of data sent between synced stores even in a distributed, peer-to-peer setting.

Updating the server's head uses optimistic locking. To update the head you need to include the last read head in your request.

3.10 Handling Conflicts

3.11 Integration with Application Logic

- demonstrate how to interface with standard MVC frameworks like Backbone, Ember.js

3.12 (Managing Changes to Distributed Logic)

The additional client logic has to be maintained and upgraded for new releases of the application. As the client logic is distributed among all users of the application, a code upgrade becomes more complex to manage than a simple server update. We will see how

the same logic used to synchronize application data can be used for updating distributed application code.

- on the server its easy - we can use a distributed version control system - they don't run on the client -> we need an app-embedded solution

4 Evaluation

We evaluate the implementation based on the set of requirements specified in section 3.2.

Evaluate the proof-of-concept by simulating syncing of data structures used in the problem scenarios with realistic network latency and disconnection.

show efficiency both on client-server and peer2peer.

Bibliography

[1] Tancred Lindholm. XML-aware data synchronization for mobile devices. 2009.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. 41(6):205–220, 2007.

[3] David Ratner, Peter Reiher, Gerald J Popek, and Geoffrey H Kuenning. Replication requirements in mobile environments. *Mobile Networks and Applications*, 6(6):525–533, 2001.

[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.