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: 17. April 2013

Abstract

Zusammenfassung

Inhaltsverzeichnis

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 Relational Data Synchronization

The Wunderlist app serves as an example for a common kind of data model that requires syncing of data in a relational schema.

Wunderlist's schema could be defined as the following:

- User (name, email, has Todo Lists)
- Invited User (name, email)
- Invited User List (has Invited Users)
- Todo Item (title, description, due date, belongs to Todo List)
- Todo List (name, belongs to Users, has Todo Items)

The User type has a singleton instance who represents the user of the app.
Users can be invited to Todo Lists. As their list of Todo Lists is hidden from the current user Invited User is a separate type.
Invited User List is simply a cached list of all users that have been invited in the past.
While Invited User List is an unordered list, Todo Lists and Todo Items are ordered.

Syncing lists of unordered object IDs never causes conflicts while syncing ordered object IDs can cause order conflicts.

3.1.2 Hierarchical Data Synchronization

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 common scenarios we derive a set of requirements for a synchronization solution.

Requirements for strategies:

- Causality preservation
- Eventual consistency
- Optimistic synchronization
- Expose conflicts
- Support peer-to-peer or hybrid synchronization
- Integration with existing app logic

(TODO: need to explain why this set of requirements, constraints on mobile devices...)

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.

Literaturverzeichnis

[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.