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

Bachelor Thesis

Data Synchronization on Mobile Devices

Name: Mirko Kiefer
Matricle: 2746040
Supervisor: Prof. Dr. Gertz
Date of Submission: 3. April 2013

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: 3. April 2013

Abstract

Zusammenfassung

Inhaltsverzeichnis

1 Introduction

This chapter contains an overview of the topic as well as the goals and contributions of your work.

1.1 Motivation

Building collaborative apps that sync data among a group of users is hard.
The widespread adoption of mobile devices with limited network access requires the offline availability of data and apps.
Some aspects of syncing are often application specific and can therefore not be solved in a generic way.
However there are recurring patterns that can be used to build application specific solutions. The goal of this thesis is to develop a syncing framework that speeds up the development of collaborative apps.

1.2 Goals of the thesis

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.

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

(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 Existing Systems

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

3.4 Evaluation of Existing Systems

Here we evaluate solutions like CouchDB, Dropbox, iCloud, git, (Parse.com) based on the requirements.

3.5 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)

3.6 Implementation

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

As syncing is state based we need to track the entire history of a database.
Every 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.
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'.

Syncing 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.

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.

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.