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

Bachelor Thesis

Offline Usage and Synchronization in Mobile Apps with HTML5

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: May 13, 2013

Abstract

Zusammenfassung

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 The Thin-Client, Fat-Server Model . . . 1

1.1.2 The Fat-Client, Fat-Server Model . . . 1

1.1.3 The Fat-Client, Thin-Server Model . . . 1

1.1.4 The Client as a Server Model . . . 2

1.2 Goals of the thesis . . . 3

1.3 Structure of the thesis . . . 4

2 Background **5**

2.1 Stream-Based Synchronization . . . 6

2.2 History-Based Synchronization . . . 7

2.3 Three-Way Merging . . . 7

2.4 Most Recent Common Ancestor . . . 8

2.5 Content Adressable Storage . . . 8

2.6 HTML5 and Offline Applications . . . 8

2.6.1 Web Storage . . . 8

2.6.2 Web SQL Database . . . 9

2.6.3 Indexed Database . . . 9

2.6.4 Cache Manifests . . . 9

3 Designing a Synchronization Framework **10**

3.1 Application Scenario: A Collaborative Task Manager . . . 10

3.1.1 User Story 1: Creating Projects . . . 10

3.1.2 User Story 2: Creating and Editing Tasks . . . 10

3.1.3 User Story 3: Commenting on Tasks . . . 11

3.1.4 User Story 3: User Workflows . . . 11

3.1.5 Data Model . . . 12

3.2 Requirements . . . 13

3.2.1 Flexible Data Model Support . . . 13

3.2.2 Optimistic Synchronization . . . 14
3.2.3 Eventual Consistency . . . 14
3.2.4 Causality Preservation and Conflicts . . . 15
3.2.5 Flexible Network Topologies . . . 16
3.2.6 Integration with Existing Application Logic . . . 16
3.3 Architecture of CouchDB . . . 17
3.3.1 Synchronization Protocol . . . 18
3.3.2 Fulfillment of Requirements . . . 18
3.4 Architecture of Histo . . . 19
3.5 Technologies used for Implementation . . . 20
3.6 Differencing and Merging of Data Models . . . 20
3.6.1 Sets . . . 20
3.6.2 Ordered Lists . . . 20
3.6.3 Ordered Sets . . . 20
3.6.4 Dictionaries . . . 20
3.6.5 Ordered Dictionaries . . . 20
3.6.6 Trees . . . 20
3.6.7 Composite Data Structures . . . 20
3.7 Storing and Commiting Changes . . . 21
3.8 Differencing Across Commits . . . 21
3.9 Synchronization Protocol . . . 21
3.10 Handling Conflicts . . . 25
3.11 Optimizations . . . 25
3.12 Integration with Application Logic . . . 25

4 Evaluation **26**
4.1 Task Manager using CouchDB . . . 26
4.2 Task Manager using Histo . . . 26
4.3 Other Backends . . . 26

Bibliography **27**

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 tablet 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.
This implies that even simple applications that are meant for single-users have to aquire collaborative features. A single-user with multiple devices is from a technical perspective effectively collaborating with itself.
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 basically 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 including all relevant aspects 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 problem statements.

- add Things app 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:

- *Offline Availability*: How can we enable the operation of a collaborative app with frequent network partition?
- *Synchronization Protocol*: How can we efficiently synchronize changed data directly between unreliably connected devices?
- *Application Integration*: 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.

Important questions which are out of scope of this thesis are:

- *Security*: How can we manage access rights and encryption in a distributed architecture?
- *Device Discovery*: How can we discover devices in a network to collaborate with?
- *Data Transmission*: How is the data propagated among devices on a technical level?

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

We will start by defining some consistent terminology used throughout the thesis:

- *Synchronization engine*: A framework that manages functional aspects of data synchronization we call a synchronization engine.
- *Node*: A node represents a single instance of an application connected to other nodes for synchronization purposes. It could be a mobile device or a server connecting devices.
- *Synchronization protocol*: The core part of a synchronization engine that describes the communication between nodes in a synchronization engine.
- *Offline*: A node is offline if it is partitioned from the network.
- *Object*: Objects can represent any kind of data like files or entities. Objects can be composed of *parts* like lines (for files) or attributes (for entities).
- *Atom*: An atom is an object that can not be divided into parts. Examples are a line in a file or literal attributes like strings or numbers in an entity.

Most synchronization algorithms can be divided into three major components.

1. *Update Detection* refers to the process of identifying changes to the data on each client. If updates are stored explicitly we categorize the algorithm as *edit-based* which applies to most *stream-based* approaches. *State based* synchronization usually run a differencing algorithm on each synchronization operation to detect updates.
2. *Update Propagation* describes the algorithm's component which implements the propagation protocol among the synchronizing clients. The design of the propagation protocol eventually defines which kinds of network topologies the algorithm can support.

3. *Reconciliation* is the final phase where the client updates are merged and conflicts identified. In a centralized scenario this part is usually carried by the server. Distributed architectures supporting peer-to-peer synchronization are much more complex as all clients have to reconcile the received updates in an eventually consistent way.

2.1 Stream-Based Synchronization

An application that tracks each edit and sends it in a stream to remote nodes follows a stream-based synchronization protocol. Stream-based synchronization is very common among real-time document editors like Google Docs.

An edit usually represents an insert or delete operation at a certain position in the text. These edit operations are broadcasted to remote nodes and then "replayed". As participating nodes can concurrently edit a document the stream of edit operations can not just be applied without modifications.
The combination of local modifications and received edit operations from a remote node requires the transformation of the remote operations in order to be correctly applied.
The family of algorithms developed to correctly transform the edit operations is described as *Operational Transformation* [1].
If some nodes are temporarily offline while continuing to edit, the correct transformation of many concurrent edit operations becomes very complex and error-prone.
A practical problem in modern user interfaces is that it is hard to correctly capture all edits made to data. If a single edit is missed the result is a fork possibly rendering all future update operations as incorrect. Packet loss due to unreliable network connections have also be taken into account which further complicates the design of a robust algorithm.
Research has therefore investigated options for data synchronization that do not require Operational Transformation.

Commutative Replicated Data Types (CRDTs) have emerged as a viable alternative for specific use cases. A recent study by Shapiro et. al presents a range of data types designed for synchronization without concurrency control [2].
CRDTs are designed in a way that all edit operations commute when applied in causal order. Due to the restrictions on supported operations on data types, CRDTs are only applicable in a narrow set of scenarios.

2.2 History-Based Synchronization

Snapshot-based methods work by tracking and relating an application's data state over time. Instead of sending a sequential stream of raw updates, each client collects additional meta-data that allows more complex reasoning about the state of each client.
A prominent example is the distributed content tracking system *git* [3] which can resolve the most complex peer-to-peer synchronization scenarios.
Git achieves this by storing the entire history of a project's database on each client. Each edit made to objects in the database is stored as a commit object and related to its ancestors.
Through the resulting commit graph each client can identify the exact subset of updates each remote node has to receive in order to be in sync.
While it sounds extremely inefficient to store the entire history of a database, git manages to do this in a very efficient way through a *Content Addressable Store* and data compression. It is not uncommon that the uncompressed form of the current state of a git project is larger than the project's entire history.

2.3 Three-Way Merging

Three-way merging describes the concept for an algorithm that performs a merge operation on two objects based on a common ancestor.
Let A be the initial state of the object and let B and C be edited versions of A. The goal is to merge B and C into a new object D.
The merge algorithm starts by identifying the differences between A and B and between A and C.
All *parts* of object B that are neither changed in B nor in C are carried over into D.
All changes to parts of the object in B that have not been changed in C are directly accepted and added to D.
If the same parts are edited both in B and C we have a merge conflict that needs to be resolved.

Three-way merging only describes the general concept but the actual algorithm will differ based on the type of objects that are merged. Text files are the most common type of object with lines seen as the *parts*. The unix program *diff3* implements a three-way merge variant for text files [4].
Most modern version control systems implement three-way merging to allow lock-free collaboration on source code. *Git* applies three-way merging not only for text files but

for entire file system trees [3].

Tancred Lindholm designed a three-way merging algorithm for XML-documents. With the *3DM* tool there is even an implementation available [5]. As XML supports the expression of a broad range of data types this is probably one of the most generic implementations.

2.4 Most Recent Common Ancestor

- describe problem with graphs
  - describe solution referring to standard algo

2.5 Content Adressable Storage

- copy on write
  - simple verification of data, free checksums
  - git as example

2.6 HTML5 and Offline Applications

HTML5 specifies a number of client-side storage options. Most are a work in process and still have to be adopted by all browser vendors. IndexedDB is most likely going to be the standard for building offline-capable web applications. Combined with Cache Manifests, HTML5 provides all the tools necessary for building offline applications.

2.6.1 Web Storage

The simplest API is the *localStorage* standard defined in the W3C's Web Storage specification [6].
It provides a key-value store accessible from JavaScript which can store string values for string keys. Most browsers currently set a storage limit of 5 MB per site. *LocalStorage* is therefore only suitable for storing small volumes of data.
Another limitation is the interface which is synchronous. As JavaScript is single-threaded, every read or write operation will block the entire application. Frequent or large-volume read/write operations can result in a bad user experience caused by a "freezing" user-interface.
LocalStorage is currently supported by all major browsers including its mobile variants.

2.6.2 Web SQL Database

A much more advanced implementation is specified by the now deprecated *Web SQL* standard [7]. It defines a relational database similar to Sqlite including SQL support.
The proposal was strongly opposed by the Mozilla Foundation who sees a SQL-based database as a bad fit for web applications [8].
The standard was therefore only implemented by Google Chrome, Safari and Opera and their mobile counterparts in Android and iOS.
Web SQL has been officially deprecated by the W3C and support by browsers is likely going to drop in the future.

2.6.3 Indexed Database

Instead of Web SQL the standard favored by the W3C and most browser vendors is *IndexedDB*.
IndexedDB defines a lower-level interface for storing key/value pairs and setting up custom indexes. The While relatively simple, the API design is generic enough to cater for implementations of more complex databases on top. It would for example be possible to implement a *Web SQL* database using *IndexedDB*.
IndexedDB supports storing large amounts of data and defines an asynchronous API.
Unfortunately the standard has not yet been implemented across all major browsers. It is currently available in Mozilla Firefox, Google Chrome and Internet Explorer. Safari support is still missing as well as support in the default Android and iOS browser.
Luckily most browsers who have not implemented IndexedDB yet, are still supporting Web SQL. There is a polyfill available that implements an IndexedDB interface using Web SQL [9]. Application developers can therefore already base their work on the IndexedDB interface while browser vendors are catching up.

2.6.4 Cache Manifests

To truely work offline, an application has to make its static resources available locally as well. The *cache manifest* defined in the HTML standard gives you the right tool [10]. It allows you to define a local cache of all application resources like HTML, CSS, JavaScript code or other static files.
Flexible policies give fine-grained control over which resources should be available offline and which need network connection.

3 Designing a Synchronization Framework

3.1 Application Scenario: A Collaborative Task Manager

Our goal is to develop a collaborative Task Manager that can still be used if disconnected from the network. We choose this scenario because we think it represents a common type of architecture and data model for mobile applications.

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

3.1.1 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

3.1.2 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

3.1.3 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!

3.1.4 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.
- 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

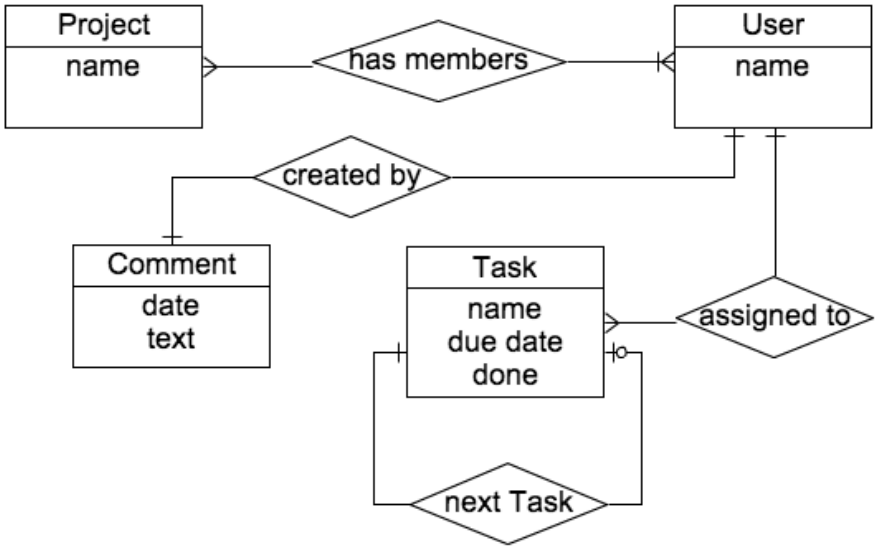

Figure 3.1: A collaborative Task Manager's data model

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.

3.1.5 Data Model

Based on the user stories we can derive a plausible data model for the application. We can map it to an 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.

3.2 Requirements

From the application scenarios we can derive a set of requirements for a synchronization solution.
The listed requirements resemble the goals set for the Bayou architecture back in 1994 [11]. Bayou had already proposed a distributed architecture with multiple devices acting as servers. At that time the computational capabilities of mobile devices were very limited. Today even smartphones have more storage and stronger CPUs than most servers in 1994. Therefore pairwise synchronization should not only be possible between servers but also between mobile devices directly.

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 level of structural awareness of an application's data. A "dumb" engine would have no awareness 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 defining requirements. The smallest pieces of information a synchronization 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 of 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 awareness of the code that is written in the

files it synchronizes. So if there are concurrent edits, git can not guarantee that merges are syntactically or semantically correct. Despite this seemingly low level of structural awareness, git is used very successfully in large software projects.

The data model of our application scenario is relatively simple but covers most of the modeling aspects the average mobile application needs:

- Entities and Instances
- (Ordered) Collections
- Attributes
- Relationships (one to one, one to many, many to many)

This set of modeling elements is represented in many client-side application frameworks like Ember.js, Backbone or Angular. If we can support synchronizing data with this type of schema it will make integration with existing frameworks fairly trivial.
We therefore require that the synchronization engine needs to have a structural awareness of at least the listed modeling components.

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.

This trade-off is enforced by the *CAP-Theorem* which states that it is impossible to have strong consistency combined with partition tolerance [12].
Most mobile applications do not require strong consistency - the offline availability of data is usually a more important factor when judging the user experience.

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 [13] 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 level 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 Flexible Network Topologies

A traveling user who works with multiple mobile devices needs to be able to sychronize data without requiring internet access. The synchronization engine should therefore be designed to handle peer-to-peer connections.
Even in an office environment where users exchange large amounts of data a direct connection can be significantly faster than doing a round-trip through a server on the internet.
For this setting a *hybrid architecture* with local servers in the company network could be an interesting alternative.
The local servers could provide fast synchronization among users inside the office while a remote server on the internet provides synchronization with users working from home.
The local and remote servers are synchronizing in a peer-to-peer topology while the users interact with them in a client-server setup.
This gives us a hierarchical architecture which is both able to exploit the different levels of network speed and guarantee a higher state of consistency through the centralized servers.

The protocol used for synchronization should be generic enough to adapt to these different network setups.

3.2.6 Integration with Existing Application Logic

Most popular operating systems for mobile devices impose restrictions on the kind of software that can be installed. Even if these limitations can be circumvented it provides a huge barrier to the install process of an app if external software is required.
For mobile applications it is therefore crucial that they can embed all their dependencies in the binary. The synchronization engine should therefore be designed as an embeddable library.
Further it is important that the interfaces are designed to be as unintrusive into the application logic as possible.

A state based synchronization strategy is required to ease the integration process. The low-level aspects of *update detection*, *update propagation* and *reconciliation* should be abstracted away from the application developer as much as possible.
At the same time the developer needs to be able to supply the logic for aspects of the synchronization that can not be solved generically. These include data model definition, conflict handling and technical aspects of messaging.

3.3 Architecture of CouchDB

CouchDB is a document-oriented database known for its data synchronization feature. It currently is a popular tool for master-less synchronization directly between devices. With multiple implementations being available for server, smartphone or in-browser deployment it seems like an excellent fit for our requirements.

The original implementation of CouchDB exposes an HTTP interface for all interactions with the database. This makes it possible to write web applications directly targeting CouchDB as the server, eleminating any middleware in between.
If an application developer is able to design his application within the constraints of CouchDB being the only backend, it is called a *CouchApp*.
CouchApps have the interesting property of being completely replicatable between CouchDB instances. So an entire working application can be deployed to a device just by replicating it from a remote CouchDB instance.
Unfortunately only few applications get by with CouchDB being the only backend required.

With *PouchDB* there has recently emerged a CouchDB implementation inside the browser in pure JavaScript. It makes use of HTML5's IndexedDB as the storage layer and can therefore be included into a web application without requiring any plugins.
PouchDB exposes a similar interface like CouchDB and can fully synchronize with an actual CouchDB instance on a server.

CouchDB's data model is relatively simple - it mainly supports the storage of JSON-documents. Each document has an ID under which it can be efficiently retrieved and updated.
There is no query language like SQL available as the stored JSON-documents are not required to have any fixed schema.
If effecient access to documents based on some of its properties is required, CouchDB allows the definition of *views*. Views are created by providing a map and possibly a reduce function. The map function is used to define an index, while the reduce function can be used to efficiently compute aggregates.

An important aspect of CouchDB is that all its operations are lockless. It achieves this by writing all data to an append-only data structure therefore never updating any data in-place. Every update of a document creates a new version of it - similar to how

some version control systems operate.
On each write of a new version, CouchDB requires that the current version ID of the document is passed. This guarantees that the client has read the current version before he is able to write any updates. If two clients concurrently update the same JSON-document, the first update that reaches the database succeeds and thereby creates a new version ID. The second concurrent update will therefore be rejected as the client did supply an outdated version ID. CouchDB treats this as an update conflict and notifies the second client. The second client can then review the changes of the first client, possibly merge it with his changes and re-send it with the correct version ID. This concept is often referred to as *Optimistic Locking*.

In the case of concurrent edits on two instances of the same database the conflict handling is more complex. Concurrent writes can no longer be linearized through optimistic locking as the two database instances are possibly disconnected.
CouchDB solves this by applying concepts of *Multi-Version Concurrency Control*. Both instances can update the same documents thereby creating two conflicting versions.
All versions of a document point to its ancestor resulting in a version tree. If the database instances synchronize each other both instances will end up with both conflicting versions of the document.
CouchDB uses a deterministic algorithm to choose one of the instances as a winner. As this choice is random to the user of an application it is often not the desired result. It is therefore possible to either pick a different conflicting version as the winner or merge both versions to a new revision.

3.3.1 Synchronization Protocol

3.3.2 Fulfillment of Requirements

As a schemaless database CouchDB at least supports the storage of any kind of data model. Its awareness of the type of data is at the same time very low.
When synchronizing databases CouchDB treats every JSON-document as an *atom*. There is no way to give CouchDB an increased level of awareness of an application's data model.
Application developers are therefore forced to write a large amount of additional merging logic inside their application.
Flexible data model support is therefore given while it requires additional app-specific logic to cater for CouchDB's lack of structural awareness.

The CouchDB model of multi-version concurrency control fulfills the requirement of optimistic synchronization.
Concurrently edited data on multiple CouchDB instances is eventually consistent if synchronized with each other.
The optimistic locking mechanism combined with a document's version tree ensures causality preservation.

CouchDB's distributed synchronization protocol supports a broad range of network topologies.

To build a more suitable solution for our requirements we can build on most of CouchDB's design decisions.
The major room for improvement lies in stronger data model awareness thereby relieving the application developer of repetitive merging logic.
We will also investigate an alternative synchronization protocol by collecting additional meta data on edits.

3.4 Architecture of Histo

Based on the requirements and the evaluation of CouchDB we derive a new 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**: synchronization does not require a central server
- **Functional Design**: only implement the functional parts of synchronization - 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**: be available on every major platform through the use of Web Standards

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 Differencing Across Commits

- Most Recent Common Ancestor algorithm used for finding common commit of clients
  - walk commit graph until MRCA
  - recursive application of MRCA on every fork in graph
  - implementation as separate module
  - use per-commit diff to find full data diff across commits

3.9 Synchronization Protocol

Synchronization always happens from a *Source* node to a *Target* node. If it is run simultaneously with Source and Target exchanged, it keeps both nodes in sync with each other.
The algorithm is designed to be able to run independently of the Source or Target. It could be implemented as a separate application possibly even running on a different device - as long as it has access to both the Source and Target node.
The Synchronizer could be run in regular intervals or explicitly triggered by changes in the Source node.

The latest commit on a node we refer to as the *head*. A node has a *master head* which refers to the version of the data considered to be 'true' by the node.
For each remote node it synchronizes with, the node keeps a *remote tracking head*.
A remote tracking head represents what the local node considers to be the current state of a remote node.

Synchronization follows a two-step protocol, step one propagates all changed data from Source to Target, step two executes a local merge operation.

Propagation

Propagation follows the following protocol:

1. Read all commit IDs since the last synced commit from Source and write them to Target.
2. Let the Target compute the common ancestor commit ID of Target's and Source's master heads.
3. Read all changed data since the common ancestor commit from Source and write to Target.
4. Set the Target's remote tracking head of Source to Source's master head.

Once these steps are executed, the Target node has the current state of Source available locally.
The Target's head still refers to the same state as the Source data has not been merged.

Listing 3.1 summarizes the protocol as pseudo-code.

```
1
2  commitIDsSource = source.getCommitIDsSince(lastSyncedCommit)
3
4  target.writeCommitIDs(commitIDsSource)
5
6  commonAncestor = target.getCommonAncestor(target.head.master, source.
       head.master)
7
8  changedData = source.getChangedDataSince(commonAncestor)
9
10 target.writeData(changedData)
```

Listing 3.1: Propagation Protocol

The functions 'getCommitIDsSince()' and 'getChangedDataSince()' are implemented as described in section 3.8.
The most recent common ancestor algorithm used in 'getCommonAncestor()' is described in section 2.4.
The internals used by 'writeData()' and the underlying commit data model are explained in section 3.7.

Merging

Even if the Source is disconnected at this stage, the Target has all the necessary information to process the merge offline:

- The Target's master head we refer to as the *master head*.
- The Target's remote tracking branch for the Source we refer to as the *Source tracking head*.

1. Compute the common ancestor of the master head and the Source tracking head. (The common ancestor could also be re-used from the propagation step.)
2. If the common ancestor equals the Source tracking head:
   The Source has not changed since the last synchronization. The master head is ahead of the Source tracking head.
   The algorithm can stop here.
3. If the common ancestor equals the master head:
   The Target has not changed since the last synchronization.
   The Source's head is ahead of Target.
   We can fast-forward the master head to the Source tracking head.
4. If the common ancestor is neither the Source tracking head nor the master head:
   Both Source and Target must have changed data since the last synchronization.
   We run a three-way merge of the common ancestor, Source tracking head and master head.
   We commit the result as the new master head.

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

Updating the Target's head uses optimistic locking. To update the head you need to include the last read head in your request. So both the fast-forward operation or the commit of a merge result can be rejected if the Target has been updated in the meantime. If this happens the Synchronizer simply has to re-run the merge algorithm.

The merging process can be described in pseudo-code as shown in figure 3.2.

```
1
2  masterHead = target.head.master
3  sourceTrackingHead = target.head.sourceID
4
5  commonAncestor = target.getCommonAncestor(masterHead,
       sourceTrackingHead)
6
7  if (commonAncestor == sourceTrackingHead) {
8    // do nothing
9
10 } else if (commonAncestor == masterHead) {
11   // fast-forward master head
12   try {
13     // when updating the head we have to pass in the previous head:
14     target.setHead(sourceTrackingHead, masterHead)
15   } catch {
16     // the master head has been updated in the meantime
17     // start over
18   }
19
20 } else {
21   commonAncestorData = target.getData(commonAncestor)
22   sourceHeadData = target.getData(sourceTrackingHead)
23   targetHeadData = target.getData(masterHead)
24
25   mergedData = three-way-merge(commonAncestorData, sourceHeadData,
         targetHeadData)
26
27   // commit object linking commit data with its ancestors:
28   commitObject = createCommit(mergedData, [masterHead,
         sourceTrackingHead])
29
30   try {
31     // when updating the head we have to pass in the previous head:
32     target.commit(commitObject, masterHead)
33   } catch {
34     // the master head has been updated in the meantime
35     // start over
36   }
37 }
```

Listing 3.2: Merging Protocol

3.10 Handling Conflicts

3.11 Optimizations

Only keep limited history.

Clients who are disconnected for too long have to fetch redundant data.

Ideal case: remember until oldest head of nodes.

3.12 Integration with Application Logic

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

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.

implement same Task Manager with different sync backends

use common web framework like Ember.js or Sencha

make a mobile app

evaluate code complexity, robustness, performance

4.1 Task Manager using CouchDB

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

describe implementation

4.2 Task Manager using Histo

4.3 Other Backends

- Sencha IO (http://www.sencha.com/products/io/) - comes with sync code

- 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.

Bibliography

[1] C A Ellis and C Sun. Operational transformation in real-time group editors: issues, algorithms, and achievements. pages 59–68, 1998.

[2] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[3] Linus Torvalds. Git. `http://git-scm.com/`.

[4] diff3. `http://www.linuxmanpages.com/man1/diff3.1.php`.

[5] Tancred Lindholm. A 3-way merging algorithm for synchronizing ordered trees—. *Master's thesis, Helsinki University of Technology*, 2001.

[6] W3C. Web storage. `http://www.w3.org/TR/webstorage/#the-localstorage-attribute`.

[7] W3C. Web sql database. `http://www.w3.org/TR/webdatabase/`.

[8] Mozilla. Beyond html5: Database apis and the road to indexeddb. `https://hacks.mozilla.org/2010/06/beyond-html5-database-apis-and-the-road-to-indexeddb/`.

[9] Parashuram Narasimhan. Indexeddb polyfill. `https://github.com/axemclion/IndexedDBShim`.

[10] WHATWG. Offline web applications. `http://www.whatwg.org/specs/web-apps/current-work/multipage/offline.html`.

[11] Alan Demers, Karin Petersen, Mike Spreitzer, D Ferry, Marvin Theimer, and Brent Welch. The Bayou architecture: Support for data sharing among mobile users. pages 2–7, 1994.

[12] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

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