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: June 5, 2013

Abstract

Zusammenfassung

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Objectives and Approach . . . 2

1.3 Structure of the thesis . . . 4

2 Background **5**

2.1 Defining Data Synchronization . . . 5

2.2 Transactions and Consistency . . . 7

2.3 Stream-Based Synchronization . . . 9

2.4 History-Based Synchronization . . . 10

2.5 Three-Way Merging . . . 10

2.6 Lowest Common Ancestor . . . 13

2.7 Content Adressable Storage . . . 15

2.8 HTML5 and Offline Applications . . . 15

2.8.1 Web Storage . . . 15

2.8.2 Web SQL Database . . . 16

2.8.3 Indexed Database . . . 16

2.8.4 Cache Manifests . . . 16

3 Designing a Synchronization Framework **17**

3.1 Application Scenario: A Collaborative Task Manager . . . 17

3.1.1 User Story 1: Creating Projects . . . 17

3.1.2 User Story 2: Creating and Editing Tasks . . . 17

3.1.3 User Story 3: Commenting on Tasks . . . 18

3.1.4 User Story 3: User Workflows . . . 18

3.1.5 Data Model . . . 19

3.2 Requirements . . . 20

3.2.1 Flexible Data Model Support . . . 20

3.2.2 Optimistic Synchronization . . . 21

3.2.3 Causality Preservation and Conflicts . . . 22

3.2.4 Flexible Network Topologies 23
3.2.5 Integration with Existing Application Logic 23
3.2.6 Cross-Platform 24
3.2.7 Other Requirements 24
3.2.8 Summary 24
3.3 Architecture of CouchDB 25
3.3.1 Synchronization Protocol 27
3.3.2 Fulfillment of Requirements 28
3.4 Architecture of Histo 28
3.5 Merging of Models 29
3.5.1 Differencing Test Case 30
3.5.2 Differencing Algorithm 32
3.5.3 Diff Merging Test Case 35
3.5.4 Diff Merging Algorithm 35
3.5.5 Patching 35
3.6 Storing and Commiting Changes 35
3.7 Differencing Across Commits 36
3.8 Synchronization Protocol 36
3.9 Handling Conflicts 39
3.10 Synchronization Topologies 39
3.11 Optimizations 40
3.12 Integration with Application Logic 40

4 Realization **41**
4.1 Technologies Used for Implementation 41
4.2 Task Manager using CouchDB 41
4.3 Task Manager using Histo 41

Bibliography **42**

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.

The traditional architecture of collaborative 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 to 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.

Rising expectations on the user experience drove developers to increasingly move application logic to the client. Initially this has only been the logic required to render 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.

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 server-centric model developers only had to main-

tain a single technology set, 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.

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 a 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 like the clients it is just another node on the network. The difference is that the server node is continuously connected to the Internet and can therefore play a useful role as a fallback.
Note that this only describes the most extreme scenario - 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 problem statements and use cases.
TODO:
- add Things app story on how hard it is
- add graphics

1.2 Objectives and Approach

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?

TODO:
- phrase problem statements - not only questions

- what is unique to our approach? (focus on practical realization with open web standards and modern tools, offline as default)

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

TODO:
- maybe add longest common subsequence

We will start this chapter by explaining the core aspects of data synchronization. After setting it in context with traditional properties of distributed databases we will present some of the popular approaches to synchronization.

Some technical background on the practicality of local data storage based on HTML5 will follow.

This will give us a solid background to develop and reason about our own data synchronization framework.

2.1 Defining Data Synchronization

Let us introduce some basic terminology and try to systematically define what data synchronization is about.

Atoms are what we define as the lowest level of data that cannot be devided into smaller parts. Every application may have a custom definition of atoms. For a file synchronizer it may be entire files, for a source code management system it may be lines in a file, for a collaborative task manager it may be literal values like strings, numbers or dates.

Atoms can be aggregated to larger structures as *objects*. A source code management system may define objects as a sequence of lines aggregated to a file. The task manager could aggregate values like strings and dates to task objects by keeping them as 'title' and 'due date' properties. Objects can themselves be aggregated further into larger objects by declaring relationships between them. File objects can be combined to directory objects, task objects into a larger structure like a project.

A collaborative application has multiple users working on different devices on a related set of data. They are either connected directly or via servers who live on the local network or the Internet. Each device, be it a user's device or a server, we define as a *node*. Nodes can be connected through various network topologies like peer-to-peer, client-server or a hierarchical architecture. In section 3.2.4 we will go into more detail about different network topologies.

The nodes of mobile users are likely to be partitioned from their network and therefore have to be able to work in *offline* mode. Therefore application data has to be available locally so that users are not blocked from using their application. Even when connected to a network it can be beneficial to maintain data locally to increase the responseness of the application.
Objects can be edited while the node is offline. 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 initial 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*.
The updates across all nodes have to be *synchronized* in order to bring all branches to the same state
The process of synchronization can be divided into three phases. Local edits first have to be identified before they can be sent to other nodes. We refer to this step as the *update detection* phase. Some applications may explicitly track each edit as its made and store the history of edit operations. This *edit-based* approach is necessary for *stream-based synchronization* which we explain in section 2.3.
If edits are not tracked directly we have to run a differencing algorithm to detect updates. This requires us to keep previous states of the data and is detailed further in section 2.4 on *history-based synchronization*.
Once updates are detected we continue with the *update propagation* phase. A stream of edit operations or the differencing output is sent to the collaborating nodes. The details will be explained in the respective sections on the stream or history based approaches.
In a final phase the received data has to be *reconceiled* with the local data on each node. Updates have to be merged and conflicts are identified. In a centralized scenario this part is usually carried out 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.

Let us review these terms through figure 2.1 and a short summary:

- **Atoms** are the literal values that can not be divided further.
- **Objects** aggregate atoms or other objects into larger structures.
- **Nodes** are the collaborating devices in an application.
- Updates of an object leads to multiple **states** which are linked in its **history**.

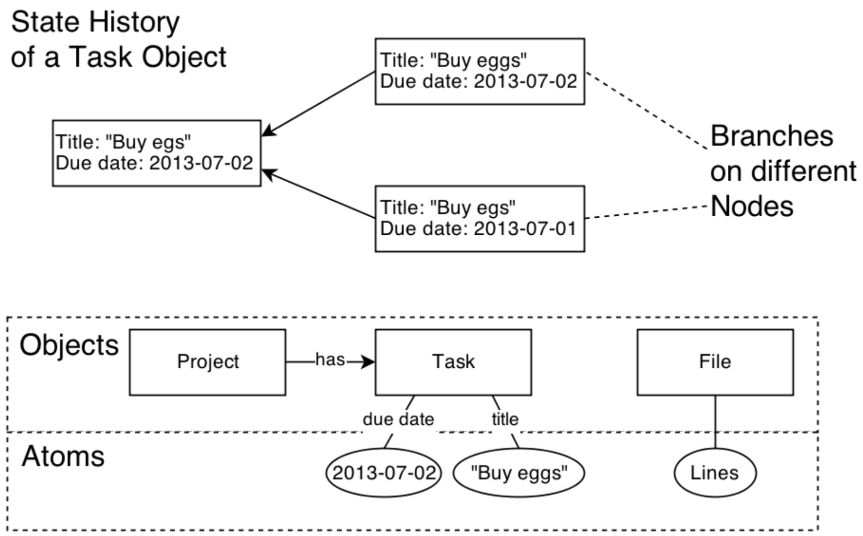

Figure 2.1: The relationship of atoms, objects, nodes and the state history

- Each object can have multiple current states across multiple nodes referred to as **branches**.
- The **update detection** phase identifies local data changes on each node.
- During **update propagation** changed data is sent to collaborating nodes.
- **Reconciliation** merges data received from other nodes and identifies conflicts.

Update detection, *update propagation* and *update reconciliation* combined are what we define as *data synchronization*.

2.2 Transactions and Consistency

The nodes of a collaborative, mobile application with replicated data represent a distributed database. Distributed database systems have been a focus of research for decades. Traditionally, the incentive to make databases distributed has been to provide fault tolerance, increase read/write throughput or to increase storage capacities. Mobile applications need data replication to reduce frequent network access resulting in a better user experience. Traditional distributed databases used to back enterprise applications running entirely in server farms. Servers are connected through reliable and high-speed networks. Network partitions are the absolute exception.
On mobile devices network partitions or slow connections are the norm. Users want to work with their notebooks even when not being in an office environment with reliable

Internet access. Mobile networks are still comparatively slow and unreliable.

Back in 1981 Jim Gray defined the properties of a reliable transaction system [1]. They are referred to as the ACID (Atomicity, Consistency, Isolation, Durability) properties - a term coined by Theo Härder and Andreas Reuter [2].
Consistency is defined as the property ensuring that a database can only transition between valid states. One way to achieve this is to use locking so that a record can not be edited concurrently. In an always-connected server environment transactions are measured in seconds - locking of data can therefore be acceptable. In a mobile setting this is not an option as transactions can easily last days. A mobile user who wants to edit some data while travelling without network connection should certainly not block all other users from doing their work. Using locking concepts in such a scenario would not only be a an inconvenience for the users but would actually lead to a high-rate of deadlocks. As Jim Gray states, the rate of deadlocks goes up with the square of the level of concurrency and the fourth power of the transaction size [3].
A common alternative to locking is *multiversion concurrency-control* (MVCC) where readers can still access the prior version of data being edited by another user [4].
With regards to the ACID property *Isolation*, MVCC refers to the most relaxed level of *"Read Uncommitted"* which allows concurrent updates and reading of uncommitted data.

Distributed databases usually use a two-phase commit protocol to gurantee strong consistency [5]. Each participant has to agree in order to successfully complete a transaction. In a mobile setting with long periods of disconnection each commit could take hours or days to be acknowledged by all nodes. This is further complicated as nodes are often not fixed and can be added or removed from a mobile application at any time. The two-phase commit algorithm requires a coordinator node which collects the votes of all collaborators and commits if the result is positive. In a peer-to-peer synchronization scenario with nodes unpredictably going offline this approach will not work as we can not define a reliable coordinator.
The *CAP-Theorem* actually states in detail that it is impossible to have strong consistency combined with partition tolerance [6].
Given these constraints we can only guarantee *eventual consistency*. Data will be consistently propagated across all nodes given a long enough period of time over which no changes are made. Eventual consistency is increasingly adopted in multi-master databases like CouchDB [7] and DynamoDB [8]. CouchDB which uses a combination of MVCC and eventual consistency will be reviewed in detail in section 3.3.
Atomicity is defined as transactions either passing entirely or leaving the database un-

changed if a part fails. When choosing eventual consistency it is clear that we can not guarantee that a transaction succeeds across all nodes. Atomicity can only be guaranteed on a per-node level with transaction results eventually being propagated.
Guaranteeing *Durability* faces the same problem with nodes not being under a centralized control. Durability on a global level can only be guaranteed if reliable server components are part of synchronization topology.

2.3 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 broadcast 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* [9].
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 [10].
CRDTs are designed in a way that all edit operations commute when applied in *causal order*. Section 3.2.3 goes into more detail about causal ordering of events. Due to the restrictions on supported operations on data types, CRDTs are only applicable in a

narrow set of scenarios.
TODO:
- more details about OT and CRDTs

2.4 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 metadata that allows more complex reasoning about the state of each client.
A prominent example is the distributed content tracking system *git* [11] 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.

2.5 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.
Figure 2.2 shows a simple scenario where a merge can be successfully made. In figure 2.3 we have concurrent edits of the same property resulting in a conflict.

There is no universal algorithm for resolving conflicts. Different types of data and applications require different types of conflict resolution strategies. In many cases conflict resolution can not even be done in an automated way but has to be left to the user of an application.
Even the term *three-way merging* only describes a 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 [12].
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 [13].
With git we have a great example of a hierarchical conflict resolution strategy:

- If two developers concurrently edit the same directory git tries to resolve this conflict by descending into the directory and looking at individual files.
- If the developers edited different files git can automatically resolve the conflict by accepting both changes.
- If the same file was edited concurrently git tries to descend a level deeper by looking at edits made to individual lines.
- If different lines were edited concurrently it can again resolve the conflict by accepting both changes.
- Only in the unlikely event that both developers edited the same line git has no way to automatically resolve the conflict. It will delegate the conflict resolution to the developers who will have to manually merge both changes.

Tancred Lindholm designed a three-way merging algorithm for XML-documents. With the *3DM* tool there is even an implementation available [14]. As XML supports the expression of a broad range of data types this is probably one of the most generic implementations.

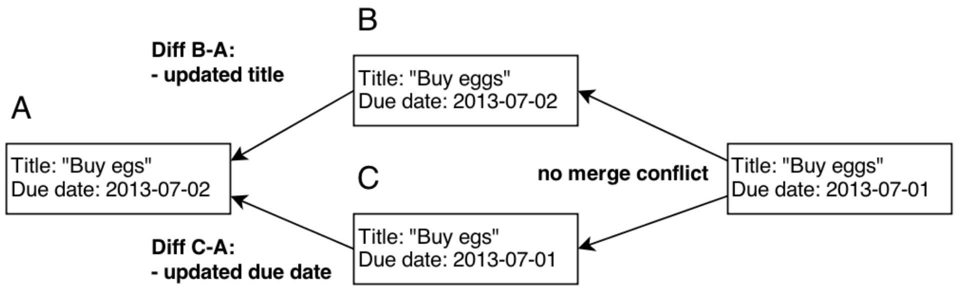

Figure 2.2: A successful three-way merge

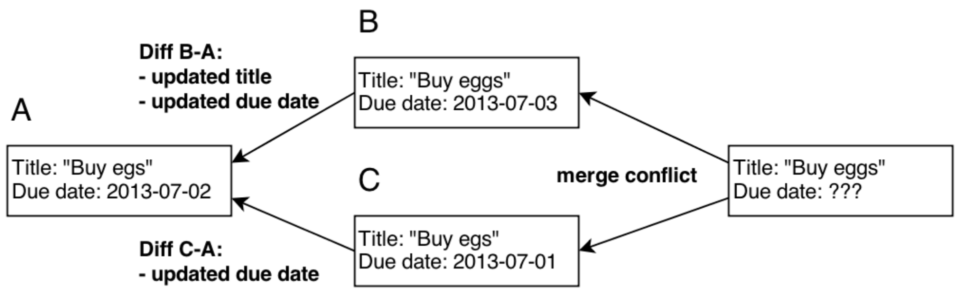

Figure 2.3: Concurrent updates of the same property result in a conflict

2.6 Lowest Common Ancestor

As described in section 2.1 the changing states of an object being updated are linked in its state *history*. Each object state links back to its ancestor thereby forming a directed acyclic graph.

The lowest common ancestor (LCA) of two states A and B in the history graph is defined as the common ancestor C with the lowest distance to A and B. The distance between two states is defined as the number of edges between them. Figure 2.4 shows an example where A and B have two common ancestors C and D but only C being the lowest common ancestor.

There are cases where the LCA is ambigous - in figure 2.5 both C and D have the same distance to A and B.

The LCA problem has long been solved through various approaches. Czumaj et al. presented a simple method solving the problem on n nodes and m edges in $\mathcal{O}(n * m)$ [15].

An alternative approach by Bender et al. is able to compute LCA queries in constant time after $\mathcal{O}(n^3)$ pre-processing step [16].

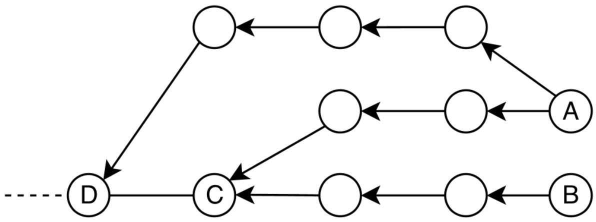

Figure 2.4: C is the lowest common ancestor of A and B

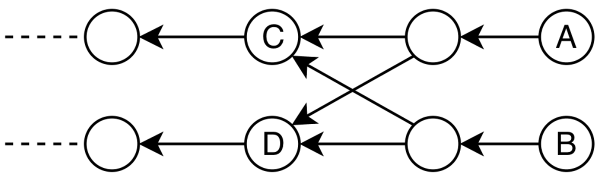

Figure 2.5: Both C and D are lowest common ancestors of A and B

2.7 Content Adressable Storage

A content adressable store (CAS) allows data to be retrieved based on its content rather than by its location. When writing an object typically a cryptographic hash function is used to compute its hash. The hash then becomes the address of the object under which it is written. It implies that data objects are always copied on write as their storage location is defined by its content. The ZFS filesystem uses this concept internally to achieve fast snapshotting and strong data verification [17].
Creating a snapshot of an object does not require any copying - the filesystem only has to keep the current version of the object as updates are made.
Data verification is given for free as well - an object can simply be re-hashed on a read and compared to the hash its stored at.
Git is another system making use of a CAS as it needs to keep all previous versions of each object [11].
Our own synchronization framework described in chapter 3.4 will use the concept of a CAS.

2.8 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.

2.8.1 Web Storage

The simplest API is the *localStorage* standard defined in the W3C's Web Storage specification [18].
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.

2.8.2 Web SQL Database

A much more advanced implementation is specified by the now deprecated *Web SQL* standard [19]. 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 [20].
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.

2.8.3 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. 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 [21]. Application developers can therefore already base their work on the IndexedDB interface while browser vendors are catching up.

2.8.4 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 developers the right tool [22]. 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

TODO:
- overview of chapter

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

TODO:
- add workflow graphic

- 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 project A 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 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 schema for the application. We can map it to an entity-relationship schema 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.

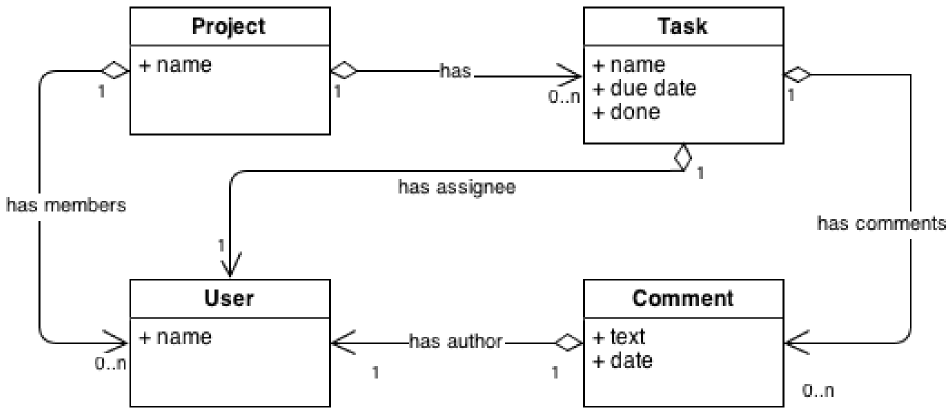

Figure 3.1: A collaborative Task Manager's data schema

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 [23]. 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 cannot 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 is composed of 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 - section 2.5 explains git's hierarchical merge strategy in more detail. By treating each line in a file as an atom git can 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 cannot 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 strong 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. Most mobile applications do not require strong consistency - the offline availability of data is usually a more important factor when judging the user experience.
Our goal is to guarantee that after a finite number of synchronization events all objects will eventually converge to the same state across all devices. This property is referred to as *eventual consistency* and explained in section 2.2.

3.2.3 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 [24] helps to solve 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. The order of states defined through the happens-before relationship is called the *causal order*. The causal order of states 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.
Merging of updates and conflict resolution should be based on three-way merging. This means each client has to keep previous versions of edited objects. The concept of three-way merging is explained in section 2.5.

3.2.4 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 robustness through the centralized servers. A hierachical architecture can counter the higher risks of failure on clients like notebooks or smartphones. The more durable and always connected server nodes should be able to recover data loss on client nodes.
The protocol used for synchronization should be generic enough to adapt to these different network setups.

3.2.5 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.

3.2.6 Cross-Platform

The application described in our scenario has to run on a multitude of devices and platforms.
Notable platforms that should be supported are:

- Desktop Operating Systems: Microsoft Windows, OSX and Linux distributions
- Mobile Operating Systems: Android, iOS
- Server Operating Systems: Linux distributions

With such a broad range of platforms it is important to target a cross-platform environment to avoid having to maintain separate implementations for each platform.

3.2.7 Other Requirements

The ideal synchronization framework would support an even broader range of requirements.
We intentionally leave out the technical aspects of data transmission between nodes. There is a broad range of protocols available to transfer data on the Internet and directly between devices. A generic synchronization solution should be agnostic to the data transmission protocol. Important is only the exposure of a data transmission interface so that adapters can be implemented for specific protocols.
Peer discovery is another important technical aspect that needs to be solved for any real application. There already exist solutions for this problems, examples being the various implementations of zeroconf [25] including Apple's Bonjour service or the competing Universal Plug n' Play (UPnP) [26] standard by Microsoft.
Security related aspects like peer authenticating and encrypted data transmission are other important requirements. Some applications need granular control on which parts of the data are transferred to which nodes, which parts are editable etc. In this thesis we assume that all collaborating users keep an entire data store in sync. Synchronizing only subsets of data with selected nodes can possibly be modeled through multiple stores each having different collaborators.

3.2.8 Summary

We can summary the defined requirements as the following:

1. **Flexible Data Models**: be able to synchronize data models including entities with attributes and relationships, entity instances and instance collections.

2. **Optimistic Synchronization**: no locking of data while ensuring eventual consistency.

3. **Causality Preservation and Conflicts**: preserve causality defined by the happened-before relationship and expose conflicts.

4. **Three-Way Merging**: base reconciliation on three-way merging concepts.

5. **Flexible Network Topologies**: design a protocol that is able to synchronize data in a peer-to-peer, client-server or hierarchical architecture.

6. **Integration**: be as unobtrusive to an application as possible.

7. **Cross-Platform**: be able to run on modern desktop and mobile platforms.

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.
There is no official documentation of CouchDB's internals - the sources used for our analysis is the CouchDB Guide written by core developers [7], the CouchDB Wiki [27] and the source code [28] for details of the replication protocol. Note that the CouchDB authors actually refer to synchronization as 'replication'.
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

CouchDB does not really have a synchronization protocol at all - it is actually a fairly simple algorithm that uses only the existing HTTP interfaces. A ‘CouchDB synchronizer’ therefore does not even have to run inside CouchDB but can be an external program that only needs access to the public interfaces of two CouchDB instances. What makes CouchDB’s synchronization work is at the core a *changes stream* that is accessible through an HTTP interface as well. Every update of a document triggers an update of the changes stream thereby adding a new entry with an update sequence ID, the new version ID of the updated document and the document ID itself. At any point the changes stream includes all updates made to the database.

The synchronization process from a CouchDB instance A to B follows the following steps:

1. Read the *last source sequence ID* stored on B - it represents the last update it read on the previous synchronization.
2. Read a few entries from the changes stream of A starting at the last source sequence ID.
3. Send B the set of document revisions - B responds with the subset of those not stored in B.
4. Fetch the missing document revisions from A and write them to B.
5. Update the last source sequence ID on B.
6. Restart the process if there are remaining updates.

All steps only use public interfaces exposed by both instances. Continuous synchronization can be implemented trivially by infinitely repeating the steps.
As explained before the process may result in conflicting document versions. It is the responsibility of the application developer to handle those conflicts after each synchronization.

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 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* and exposure of *conflicts*.
CouchDB's distributed synchronization protocol supports flexible network topologies.
Cross-platform support is given with implementations available for servers, mobile devices and even web browsers.

To build a more suitable solution for our requirements we can build on many of CouchDB's design decisions.
Major room for improvement lies in stronger data model awareness thereby relieving the application developer of repetitive logic and improving *unobtrusive integration* into an application.
CouchDB only remembers the version history of a document in the form of version IDs. The actual documents are not retained not even those of common ancestors in the case of conflicts. CouchDB's version history can therefore only be used to identify conflicts but does not support *three-way merging*.
TODO:
- Contrast synchronization with the way git works

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**: history-based 3-way merging

- **No Change Tracing**: explicit 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 Merging of Models

In this section we will focus on the core merging semantics leaving out details about update propagation which will be discussed later in section 3.8.
As described in section 2.1 synchronization always starts with an update detection phase. Before we can start to merge branches we need to know about what has changed. One our our design goals is to relieve the application developer from manual change tracing. We therefore need to implement a differencing algorithm for the types of models described in section 3.2.1.
The meta model definition in the form of entities, their attributes and relationships is usually static and does not have to be merged. We have to focus on merging the actual data which is structured through entity instances, their attribute values and relationships to other instances. Most modern web application frameworks realize one-to-one relationships by simply having an attribute storing the related instance's ID. One-to-many relationships are realized through an attribute having a collection of instance IDs.
Instance collections often have a relevant order that needs to be preserved. An example is the list of tasks in a project that is displayed to the user. The user wants to be able to change the order of tasks and the order should be persisted.
Summarizing the kind of structures we need to synchronize:

- Instances with their attribute values
- Attribute values that can be literals or instance IDs (modeling one-to-one relations)

- Attribute values that can be ordered collections (modeling one-to-many relations)

We have described three-way merging semantics in section 2.5. Our algorithm will be structured into three distinct phases. Starting with a differencing phase we identify the changes made in two branches of our state B and C since our common ancestor state A. Should an application decide to explicitly track changes it could actually leave this phase out and still benefit from the rest of our merging algorithm.
If follows a diff merging phase where the two diff results A-B and A-C are combined into one diff. As its only input is the two diffs it does not require access to the actual states A, B or C. In this phase we can have conflicts if the two branches contain updates to the same parts of the state.
The merged diff can then be applied to the origin state A in order to create the actual merged state. For this step we require a patch algorithm.
Let us summarize the tree steps of the complete merge process of two branches:

- **Differencing**: we diff branch B and C with their common ancestor A.
- **Merging**: we merge the two diffs A-B and A-C to a new diff result.
- **Patching**: we apply the merged diffs as a patch to A which results in the merged state D.

3.5.1 Differencing Test Case

To show correctness of a differencing algorithm we need to define sample model states with expected differencing results. Based on an ancestor state all users start with we define several possible branch states. For each branch state we will define the difference to the ancestor state. This set of data can then be used as a test case for our implementation.

Ancestor state A of Projects

ID	Project Name	Members	Tasks
1	Marketng Material	Rita, Tom, Allen	1, 2, 3, 4
2	Product Roadmap	Rita, Allen	5

'Members' is an unordered collection of users - we define the user's IDs identical to their names.
'Tasks' is an ordered collection of tasks IDs.
We will leave out the details about other entities as the project instances already cover

all required modeling aspects.

State B of Projects

ID	Project Name	Members	Tasks
1	Marketing Material	Rita, Tom	1, 4, 2, 3, 6
2	Product Planning	Rita, Allen	5

State C of Projects

ID	Project Name	Members	Tasks
1	Marketing Strategy	Rita, Tom, Allen	4, 1, 2, 3
2	Product Strategy	Rita, Allen	5, 7

The respective differencing outputs of A-B and A-C are defined as the following. Note that all index positions are seen as relative to the origin state. So even if one diff contains multiple insert, move or remove operations they are given as if they were all applied simultaneously to the origin state.

Expected diff A-B

ID	Project Name	Members	Tasks
1	insert ‘i’ behind index 5	remove ‘Allen’	move index 3 behind index 0 insert 6 behind index 3
2	remove from index 8 to 15 insert ‘Planning’ behind index 7	unchanged	unchanged

Expected diff A-C:

ID	Project Name	Members	Tasks
1	insert ‘i’ behind index 5 remove from index 9 to 17 insert ‘Strategy’ behind index 8	unchanged	move index 3 behind index -1
2	remove from index 8 to 15 insert ‘Strategy’ behind index 7	unchanged	insert 7 behind index 0

In this differencing output we actually do not treat the project name as an atom but actually go a level deeper and difference its characters.
Depending on the application this might not be necessary. A simpler output which treats the project titles as atoms would look like this:

Simpler 'Project Name' diff A-B

ID	Project Name
1	change to 'Marketing Material'
2	change to 'Product Planning'

Simpler 'Project Name' diff A-C

ID	Project Name
1	change to 'Marketing Strategy'
2	change to 'Product Strategy'

The choice of differencing granularity will affect what we will see as conflicts in the merging stage.

3.5.2 Differencing Algorithm

Differencing algorithms have been studied extensively. There exist efficient solutions for a range of data structures. It is not a focus of our thesis to develop the most efficient differencing algorithm matching our scenario. Our goal in this section is to show the practical feasibility of the three-way merging component in our architecture. We favor a simple solution, re-using existing concepts so that we can broaden our focus on an other areas of our synchronization solution.
The test case defined in the previous section can be decomposed and mapped to common data structures.
An instance, in our example a project, actually represents a *dictionary* data structure or a set of key-value entries. The dictionary keys are mapped to the attribute names and the dictionary values to the respective attribute value.
Attribute values that represent collections of instance IDs can be mapped to *sets*. Our 'Members' attribute does not have a relevant order, it can be mapped to a normal set. The 'Tasks' attribute has a significant order and therefore has to be mapped to an *ordered set*.
If we do not treat string values as atoms we have to represent them as an *ordered list*.

To summarize - for the following data structures we need an algorithm that finds the difference from an origin state A to a changed state B:

- **Dictionaries**: to model entity instances.

- **Sets and Ordered Sets**: to model instance relationships.
- **Ordered Lists / Strings**: to model string values of instance attributes.

Ordered list or string data structures have been the main focus in previous research on difference algorithms. Myers presented an efficient algorithm with $\mathcal{O}(n * d)$ time and space complexity to difference two strings A and B with n representing the sum of the lengths of two strings and d the size of the shortest edit script transforming A to B [29]. The shortest edit script is equivalent to the result of a differencing algorithm. As in practical applications differences are usually small the algorithm performs well.
Taking the ordered list difference algorithm as given, it can easily be re-used to build an algorithm for ordered sets. Ordered lists can only have differences in the form of *insert* or *remove* operations. Ordered sets extend this - the simultaneous remove and insert of a globally unique element is now considered as a move operation.

To implement an ordered set algorithm we can therefore take the output of the ordere list algorithm and scan the result for the remove and insert of the same element. This can be efficiently implemented through a hash:

1. Scan the diff result and build a hash for all removed elements.
2. Scan the result again and test for all inserted elements whether they are included in the hash.
3. If a match is found replace the remove and insert operations through a single move operation in the result.

The time complexity of building a hash can be estimated with $\mathcal{O}(nlog(n))$ with n representing set size. The match searching has linear time complexity. We therefore only add $\mathcal{O}(nlog(n))$ complexity to Myers difference algorithm for a naive solution for ordered sets.

For simple sets we can implement a naive solution through two hashs of the respective set entries combined with a scan through each set:

1. Add all entries of set A to a hash H_A and those of set B to a different hash H_B.
2. Scan set A and test for matches in hash H_B.
3. If no match is found add a remove operation to the result.
4. Scan set B and test for matches in hash H_A.

5. If no match is found add an insert operation to the result.

Insert and *remove* are the only operations in a set.
The time complexity for building each hash is again $\mathcal{O}(nlog(n))$. Searching for matches has linear complexity which results in a time complexity $\mathcal{O}(nlog(n))$ for the entire algorithm.

A dictionary data structure has *insert*, *remove* and *update* operations. If the instance described through the dictionary has a fixed set of attributes it would actually only need to support an update operation. In modern web applications it is not uncommon that there is no fixed data schema. It is often the case that new attributes are added to instances at runtime. Even if there is a fixed schema it might be changed through a software update with old instances not being migrated to the new schema. We should therefore support the full set of dictionary operations in our difference algorithm.
A simple and efficient solution is to again use two hashes for fast lookup combined with a scan through both dictionaries:

1. All all keys and values of dictionary A to hash H_A and those of dictionary B to hash H_B.
2. Scan through all key-value entries of A.
3. If the key is not included in hash H_B, add a remove operation to the result.
4. If the key is included and the value in H_B is different, add an update operation.
5. Scan through all key-value entries of B.
6. If the key is not included in hash H_A, add an insert operation to the result.

Building the hash is again estimated with time complexity $\mathcal{O}(nlog(n))$, scanning both dictionaries has linear time complexity. The combined time complexity is therefore again $\mathcal{O}(nlog(n))$.
Depending on the application, its instances are often already implemented with hash-like lookup performance - in this case we could skip step 1. The time complexity is in this case only linear.

Differencing of our entire data is actually a hierachical process combining all of these algorithms.
At the highest level we have a set of project IDs where we use our set difference algorithm

to identify removed or new projects. For each project we then need to dive a level deeper and difference the actual instances - here we use the dictionary algorithm. If our project only has atoms as attribute values the process stops here. If there are string values we might choose to do finer grained differencing using the respective algorithm. If we have one-to-many relationships in the instance we have to compute an (ordered) set difference.
The same process has to be repeated for each entity's instances.

3.5.3 Diff Merging Test Case

3.5.4 Diff Merging Algorithm

3.5.5 Patching

TODO:
- 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
- efficient child tree pointers like in git
- instances are key-value sets
- collections are ordered sets
- take ordered-list diff as granted

3.6 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.

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

3.7 Differencing Across Commits

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

3.8 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.

```

commitIDsSource = source.getCommitIDsSince(lastSyncedCommit)

target.writeCommitIDs(commitIDsSource)

commonAncestor = target.getCommonAncestor(target.head.master, source.
    head.master)

changedData = source.getChangedDataSince(commonAncestor)

target.writeData(changedData)
```

Listing 3.1: Propagation Protocol

The functions 'getCommitIDsSince()' and 'getChangedDataSince()' are implemented as described in section 3.7.
The most recent common ancestor algorithm used in 'getCommonAncestor()' is described in section **??**.
The internals used by 'writeData()' and the underlying commit data model are explained in section 3.6.

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.

```

masterHead = target.head.master
sourceTrackingHead = target.head.sourceID

commonAncestor = target.getCommonAncestor(masterHead,
    sourceTrackingHead)

if (commonAncestor == sourceTrackingHead) {
  // do nothing

} else if (commonAncestor == masterHead) {
  // fast-forward master head
  try {
```

```
    // when updating the head we have to pass in the previous head:
    target.setHead(sourceTrackingHead, masterHead)
  } catch {
    // the master head has been updated in the meantime
    // start over
  }

} else {
  commonAncestorData = target.getData(commonAncestor)
  sourceHeadData = target.getData(sourceTrackingHead)
  targetHeadData = target.getData(masterHead)

  mergedData = three-way-merge(commonAncestorData, sourceHeadData,
      targetHeadData)

  // commit object linking commit data with its ancestors:
  commitObject = createCommit(mergedData, [masterHead,
      sourceTrackingHead])

  try {
    // when updating the head we have to pass in the previous head:
    target.commit(commitObject, masterHead)
  } catch {
    // the master head has been updated in the meantime
    // start over
  }
}
```

Listing 3.2: Merging Protocol

3.9 Handling Conflicts

TODO:
- application specific, no general solution
- automatic resolution strategies
- manual resolution through user

3.10 Synchronization Topologies

TODO:
- document different supported topologies

- client-server
- client-client
- client-server + server-server
- hierarchical (office server + cloud server)

3.11 Optimizations

TODO:
- 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

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

4 Realization

Realize a proof-of-concept and simulate syncing of data structures used in the problem scenarios with realistic network latency and disconnection.
TODO:
show efficiency both on client-server and peer2peer.
implement same Task Manager with different sync backends
–> evaluate ease of integration with app logic
use common web framework like Ember.js/Angular
evaluate code complexity, robustness, performance

4.1 Technologies Used for Implementation

We describe implementation details like the technologies used, code structure and the testing framework to evaluate the system.
TODO:
- everything web-based –> only way to be cross-platform
- client-side persistence with HTML5
- note on alternatives (Lua, native)

4.2 Task Manager using CouchDB

describe implementation

4.3 Task Manager using Histo

describe implementation

Bibliography

[1] Jim Gray. The transaction concept: Virtues and limitations. pages 144–154, 1981.

[2] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[3] J Gray, P Homan, H Korth, and R Obermarck. A strawman analysis of deadlock frequency. *in SIGOPS Review*, 1981.

[4] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[5] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[6] 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.

[7] Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2010.

[8] 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.

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

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

[11] Travis Swicegood. *Pragmatic version control using Git*. Pragmatic Bookshelf, 2008.

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

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

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

[15] Artur Czumaj, Mirosław Kowaluk, and Andrzej Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1-2):37–46, June 2007.

[16] Michael A Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, November 2005.

[17] Roman Strobl and OpenSolaris Evangelist. Zfs: Revolution in file systems. *Sun Tech Days*, 2009:2008, 2008.

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

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

[20] 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/`.

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

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

[23] 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.

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

[25] Mesut Günes and Jörg Reibel. An ip address configuration algorithm for zeroconf mobile multihop ad hoc networks. In *Proceedings of the International Workshop on Broadband Wireless Ad-Hoc Networks and Services*. Citeseer, 2002.

[26] Brent A Miller, Toby Nixon, Charlie Tai, and Mark D Wood. Home networking with universal plug and play. *Communications Magazine, IEEE*, 39(12):104–109, 2001.

[27] Apache couchdb wiki. `http://wiki.apache.org/couchdb`.

[28] Couchdb source code. `https://github.com/apache/couchdb`.

[29] Eugene W Myers. An O (ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.