Building advanced, offline web applications with HTML 5

Robin Leblon

Promotors	prof. dr. ir. Frank Gielen (Ghent University)
	prof. dr. Albert Oliveras i Llunell (UPC-Barcelona Tech)
Co-Promotor	prof. dr. Enric Rodríguez Carbonell (UPC-Barcelona Tech)

Master thesis made to obtain the academic degree of Master of Science : Computer Sciences (Software Engineering)

UPC-Barcelona Tech
Faculty of Informatics
Department of Languages and Informatics Systems
Chairman : Ana Ibañez Julia
Academic year 2009-2010

Ghent University
Faculty of Engineering Science
Department of Information Technology
Chairman : prof. dr. ir. Daniël De Zutter
Academic year 2009-2010

Building advanced, offline web applications with HTML 5

Robin Leblon

Promotors	prof. dr. ir. Frank Gielen (Ghent University)
	prof. dr. Albert Oliveras i Llunell (UPC-Barcelona Tech)
Co-Promotor	prof. dr. Enric Rodríguez Carbonell (UPC-Barcelona Tech)

Master thesis made to obtain the academic degree of Master of Science : Computer Sciences (Software Engineering)

UPC-Barcelona Tech
Faculty of Informatics
Department of Languages and Informatics Systems
Chairman : Ana Ibañez Julia
Academic year 2009-2010

Ghent University
Faculty of Engineering Science
Department of Information Technology
Chairman : prof. dr. ir. Daniël De Zutter
Academic year 2009-2010

Preface

PERMISSION FOR USE OF CONTENT

[illegible]

- Robin Leblon
June 2010, Barcelona

UNIVERSITEIT GENT

UPC

ABSTRACT

Purpose
This project tackles two typical problems of *web applications*: they (1) suffer a noticeable delay when fetching remote data, and (2) are unavailable when no working internet connection is available.

Design/methodology/approach
Firstly, a range of recently developed specifications (mostly part of *HTML5*), which aim to provide *client-side storage* by using a local browser database, are critically compared and tested.
Secondly, we look into distributed *database replication* in general. After carefully choosing the required replication features, we select the one that suits our purposes the best.
Ultimately, these two concepts are united in a quality attribute driven software architecture for web developers.

Result
An extendable, publicly available library architecture is described, using industry-standard design patterns in up to three iterations, and implemented.

Originality/value
Although the concepts of database replication and client-side storage are widely researched and employed, our state of the art study showed that no project has united them. Moreover, there appears to be serious interest in transparently synchronized browser databases in the web development world.

Future work
The present project provides a solid foundation to build a more robust solution, incorporating other important aspects of web applications : i.e. security, performance, integration with existing web frameworks, etc.

Keywords : *Web applications, HTML5, Client-side storage, Browser databases, Database replication/synchronization*

UNIVERSITEIT GENT

UPC

ACKNOWLEDGEMENTS

This dissertation could not have been written without prof. dr. Albert Oliveras i Llunell and prof. dr. Enric Rodríguez Carbonell from UPC-Barcelona Tech, who both served as my supervisors. They have guided me through the dissertation process, never accepting less than my best efforts.
I thank them both.

Furthermore, I would like to thank prof. dr. ir. Frank Gielen for providing support from Ghent University.

The people of crossbrowsertesting.com, who have made the development and testing process a great deal easier by allowing me to use their automated web-testing services free of charge. Thank you.

Most especially I would like to extend my gratitude to my family, parents and friends, for the constant reminders and much-appreciated motivation, not only during this project, but throughout my entire education.

- Robin Leblon
June 2010, Barcelona

INDEX

Chapter 1

Introduction

PROJECT MOTIVATION

The problem

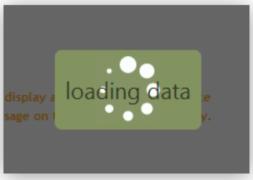
Imagine an application that ...

- feels like a *native* application
- looks like a *native* application
- performs like a *native* application

... but is actually a **web** application.

With the recent breakthrough of AJAX[1] technologies, websites have transformed from simple static content delivery into full featured, interactive applications. New client-side user interface (UI) libraries (e.g. JQuery UI) have made it possible to create a uniform look throughout a web application with advanced UI elements like draggable widgets, dialogs, sliders, progress bars, date- and color-pickers or rich animation effects.

The look and feel of web applications has come a long way, and might even be on par with native applications. However, when it comes to **performance**, it is a different story: no matter how much asynchronous loading is performed in a web application, when a remote database is queried, it always results in a noticeable delay.



The result of this delay is that the user is constantly reminded that he is, in fact, using a web page and not a native application.

The solution : HTML5 ?

Ever since the introduction of HTML 4.01 in December 1999 [1], development of its successor has slowed down significantly. The World Wide Web Consortium (W3C) shifted its focus more towards XHTML as the successor for HTML 4.01, which led to the creation of the WHATWG[2] in 2004.
The WHATWG's main focus was -and still is- on the development of HTML 5, and they have made great progress over the last years. So much, in fact, that the W3C has decided to adopt their HTML 5 specification proposal as the official successor to both XHTML 1.1 and HTML 4.01.

The current version of HTML 5 is geared mostly towards developing Rich Internet Applications (RIA's) more easily, by adding native video, audio, 3D, drawing, etc. support to the browser. However, another interesting new feature is the addition of a **local database** to the browser.

1 Asynchronous Javascript And XMLHttpRequest | see glossary for more information on this subject

2 Web Hypertext Application Technology Working Group | see glossary for more information on this subject

The goal

As introduced in the previous section, the HTML 5 specification includes a way to save data on the client's computer. More precisely, it allows a full-featured database to be included in the browser, allowing developers to persist data for offline use. This database can then be queried using an SQL-dialect, and has full support for ACID transactions.

Now imagine we copy all relevant data from the remote database to the local browser database. We then replace the AJAX calls (to load remote data) by queries on the local database, and pass this on to the web page. If we manage to keep the data in the local database and the remote database synchronized, we have solved the web application performance problem.

This is precisely what we are trying to achieve : specify, design and implement a library to allow web application developers to work more easily with client-side, browser databases.

In addition to improving the performance of web applications, we also offer a solution for another problem. Specifically, local browser databases allow us to *use the web application without an internet connection*.

CONTENTS

Chapter 2 acquaints the reader with the most important web technologies needed for this project, and the differences between them.

Chapter 3 presents the state of the art. The implementations of the technologies introduced in chapter 2 are tested and evaluated. Every topic ends with a well-founded conclusion deciding whether the featured algorithm/technology/concept is useful for this project and a concise choice of which one will be used and to what end.

Chapter 4 introduces the project vision. The key features are listed and a rundown of quality attributes summarizes the primary objectives.

Chapter 5 contains use case scenarios for both end-users and developers using this project in their web application. Next, it describes how the quality attributes introduced in the project vision will be achieved throughout the project, by use of quality scenarios.

Chapter 6 gathers all the knowledge from the previous chapters to define an architecture, featuring all the characteristics mentioned in the project vision, while living up to the quality scenarios defined in chapter 5. The project architecture is described in fine detail, and is illustrated by numerous sequence scenarios. Lastly, a deployment diagram is provided.

Chapter 7 shows the project planning. Additionally an estimate is made of the total cost of this project.

Chapter 8 expands beyond the scope of this project to explore future work and describes the issues that have to be worked out for this project to be a viable solution for real-life web applications.

Following these chapters are a **glossary** of terms and a **reference listing** of all sources used throughout the text.

Chapter 2

Essential technologies

INTRODUCTION

To be able to achieve this project's goals, we will be using some cutting-edge web technologies. The reader will be acquainted with these technologies in this chapter. Each technology is introduced, an overview is given and its features are evaluated and compared to competing alternatives.

LOCAL RELATIONAL DATA STORAGE

As stated in the project motivation, the incentive for this project started out with the HTML 5 **Web SQL Database** specification. However, after a quick test of the most popular browsers, there appears to be an alternative for storing data client-side which has been adopted by some major browser developers (e.g. Mozilla), namely the **Indexed Database** specification.

In this chapter, both of these specifications will be explored and their differences explained. Additionally a slightly older way of saving data locally, **Google Gears**, will be introduced and we will explain how Web SQL Database and Indexed Database stack up against it. It has been included for backwards-compatibility purposes, as the two new specifications are still immature and unfinished.

Web SQL Database | Introduction

From the W3C's latest published Web SQL Database specification :

> *"This specification defines an API for storing data in databases that can be queried using a variant of SQL."*

A more elaborate definition would incorporate the fact that this database is stored locally, on the client's computer, and can be queried using an SQL dialect in JavaScript[3].

The specification itself is straightforward, developer-oriented and contains examples. For a comprehensive study we direct the reader to :
`http://www.w3.org/TR/webdatabase`

To be able to understand the rest of this dissertation, the following overview of the Web SQL Database specification should suffice.

[3] The term "JavaScript" is used to refer to ECMA262, rather than the official term ECMAScript [2], since the term JavaScript is more widely known and accepted.

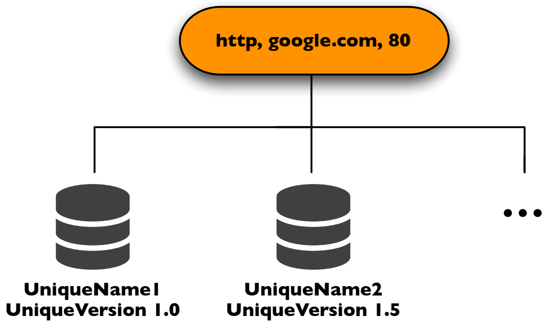
Origin

An *origin* is defined by the W3C as the tuple : (scheme, host, port) [3]
Examples of valid origins : `(http, google.com, 80)`
`(ftp, be.ubuntu.com, 2121)`

The definition also implies that a subdomain is a different origin, however a web path is not. As it will become clear later, this has some important consequences.

Databases

Every origin is associated with a set of databases. Each database has a unique name, and exactly one version; a database cannot exist in multiple versions at the same time.
The advantage of this is that developers are ensured that no new code is run on an older database scheme, while also allowing them to manage schema changes incrementally and non-destructively.



Since applications on the same domain, but on a different path have the same origin, these could access each others' databases. This could pose a problem for applications running on big shared domains. Thus, web applications using this technology are to be run on different subdomains instead of on different paths. For large, performance-minded applications, the incurred overhead does not appear to be a big sacrifice.

A 5 MB quota per origin is recommended by the W3C, with the user being prompted when the quota is about to be exceeded. The API also allows the developer to specify a predicted database size to improve the user experience; i.e. the browser can then ask the user if origin *X* is allowed to use *YY* MB or even GB's of space, instead of prompting the user every 5 MB increase.

A note on asynchronous programming

In browsers, JavaScript and the web page's rendering engine typically share a single thread. This means that when we perform an expensive operation in JavaScript, it might freeze the page rendering or make the page feel unresponsive, until the operation is finished.
To solve this problem, many APIs in JavaScript are implemented asynchronously. This way they do not block the page rendering process; instead the "expensive" JavaScript operation is scheduled to run in the background. When the operation finishes, a callback function is usually executed so that the rest of the script can be run after the expensive operation has been finished.

Asynchronous transactions

The Web SQL Databases API defines an asynchronous transaction[4] on an opened database with support for preprocessed statements. A statement is created with placeholders for the arguments, after which the arguments are passed through, checked (to prevent SQL injections and cross-site scripting), and the transaction executed. Although "raw" SQL statements are possible, the W3C highly recommends not to use them.
A callback function can be assigned to the transaction to allow the developer to verify if the transaction was performed successfully and/or properly respond to failure. Single SQL statements cannot be executed outside the scope of a transaction object.

Synchronous transactions

Recently, a synchronous transaction has been added to the specification. However no implementations have been found in browsers at the time of writing. Furthermore this is of no interest for this project and is only included for the sake of completeness.

Indexed Database | Introduction

From the W3C's latest published Indexed Database specification [4] :

> *"This specification defines APIs for a database of records holding simple values and hierarchical objects. Each record consists of a* ***key*** *and some* ***value****. Moreover, the database maintains indexes over records it stores."*

Furthermore, the W3C also notes that a query language can be layered on this API, since records can be accessed by their key and/or their index.

As is typical for a W3C specification, it is very well documented, highly detailed and with a lot of examples. For an in-depth view on the specification, we direct the reader to :
`http://www.w3.org/TR/IndexedDB`

For the purpose of this dissertation the following description of the specification should be adequate.

[4] A database transaction comprises a unit of work performed on a database, and treated in a coherent and reliable way independent of other transactions. A database transaction, by definition, must be Atomic, Consistent, Isolated and Durable (ACID). More information on this subject can be found in the section "Database Replication", later on in this chapter.

"Databases"

One might wonder why the title of this subsection has quotes around the title. Although in the API it is referred to as a "database" by the W3C, it is not a database in the way we normally think of databases. The records are not accessible directly, and the data cannot be queried using an SQL-like language. This important distinction has a lot of ramifications, as the reader will notice in this section's conclusion.

As in the Web SQL Database specification, each origin[5] has an associated set of databases. Every database has a unique name, a human readable description and exactly one version; a database cannot exist in multiple versions at the same time.

A database comprises:

- one or more *object stores* and
- any number of *indexes*.

These are referred to as the database's *objects*.

Indexes and object stores

An *object store* is a persistent storage object, comprised of key-value pairs, referred to as *records*. To enable fast insertions and lookups, the records are sorted by their key in the object store.
Object stores are identified by a *name*, and have a key generator which generates unique keys in a monotonically increasing sequence. This can be compared to the "Auto Key"-feature used by many DataBase Management Systems (DBMS).
Alternatively, the developer (or application) can define the keys, provided they are all unique and comparable, with total ordering. Furthermore, concurrent reading of records is supported.

Transactions

Transactions are supported in this API, although they have a slightly different implementation than one would typically expect in a classical database setup. A transaction on an Indexed Database represents an atomic and durable set of data access and mutation operations. Although transactions do offer some protection from application and system failures, they do not guarantee the ACID[6] properties as one would expect in a typical database.

Synchronous and asynchronous

All features of the Indexed Database API are both available synchronous and asynchronous.

[5] See the previous section on the Web SQL Database specification for more info on "origins"

[6] Atomicity, consistency, isolation, durability | see glossary for more information on this subject

Google Gears | Introduction

A few years ago, Google started working on a cross-browser plugin to support more powerful web applications. To accomplish this, features have been added to all mainstream browsers. Amongst these numerous features is a local browser database (SQLite) which can be accessed from JavaScript.

Although they have recently stopped its development [5] in favor of HTML5 and a more standards-based approach, the importance of Gears is not to be overlooked. Google Gears gives us a unique opportunity to bridge the gap between very new, state-of-the-art browsers implementing any of the previously mentioned specifications, and older browsers which only support cookies or some slightly more advanced key/value storage.

The Google Gears Database API is rather simple compared to the two previous APIs. For a full, detailed description including examples one can visit :
`http://code.google.com/apis/gears/api_database.html`

To acquaint the reader with the most notable features of the Google Gears Database API we have included the following section.

Google Gears | Overview

Databases

As with the two previous storage APIs, databases are stored based on their origin[7].
A database is defined solely by its name; no versioning is available. To create or access a database, the user's permission is required to verify if the origin is allowed to do so.

This API is built on a slightly modified SQLite database, with some features disabled. For example the `ATTACH` command to directly access an SQLite database file on a specified location on the user's disc, is disabled for obvious reasons. None of the purely database-specific aspects have been removed though.

Gears includes an SQLite extension called fts[8] (Full-Text Search), which allows the developer to create a table and search for words in TEXT columns. This is a very powerful extension, which allows some unique features not found in most DBMS's.

Transactions

The Gears database API does **not** support transactions.
To prevent SQL injection attacks, SQL queries are built in a similar way as in the Web SQL Database API, i.e. using a placeholder that sanitizes input.

[7] See the previous section on the Web SQL Database specification for more info on "origins"

[8] For more information on this subject we refer the reader to `http://www.sqlite.org/cvstrac/wiki?p=FtsTwo`

Synchronous

The API is completely **synchronous**.
However, another feature of Gears is the addition of a Worker Pool. The Worker Pool API allows web applications to run JavaScript code in the background, without blocking the main page's script execution. Basically, it allows one to run synchronous functions asynchronously.

Supported browsers/platforms

Google Gears runs on the following platforms/browsers.

	Windows XP, Vista or 7	Windows Mobile 5 and 6	Mac OS 10.4+	Linux 32-bit only
Internet Explorer 6+	✓			
Internet Explorer Mobile 4.01+		✓		
Opera Mobile 9.5+		✓		
Firefox 1.5+	✓			✓
Safari 3.1.1 *	✓		✓	

* all Safari 3 versions from 3.1.1 on are supported; however Safari 4 is no longer supported
+ indicates the mentioned version is supported, as well as all versions above

Conclusion

After introducing the three most important ways to save data locally, the following question naturally arises : "Which one suits this project best?". To be able to answer this question, let us start by reviewing the most important features of each API.

	SQL support	Transactions	Versioning	Asynchronous
Web SQL Database	✓	✓	✓	✓
Google Gears	✓			
Indexed Database		✓	✓	✓

Web SQL Database
Purely based on the features, it is obvious that the Web SQL Database is the most complete specification, from this project's point of view.

Though it has to be noted that it is not mature yet, nor final. However, the W3C has indicated that no drastic changes will be made to the specification when promoting it to the next stage [6].

Google Gears Database

When looking at the features of the Gears API in the table above, one might deem the Gears API severely lacking in features for this project. However, the most important feature - a query-able database - is present.

Versioning can be implemented quite easily by, for example, adding a table to the database to store the database's current version. When opening the database, the version table can be checked first and when necessary, the database updated.

The lack of *transaction support* seems like a bigger problem. However, SQLite has basic support for transactions built in [7], so a solution can be built quite easily.

Asynchronous behavior can be achieved by using an asynchronous worker from the Gears Worker Pool API for the database interactions, as pointed out above.

In the end, most of the negative aspects of the Gears Database API that are relevant for this project, can be overcome quite easily. Further, Gears allows us to provide the users without state-of-the-art browsers an alternative, albeit they might have to install a plugin.

Indexed Database

Although versioning, transactions and both synchronous and asynchronous operation is supported, the lack of being able to query the database poses a major problem.

The W3C indicates that it is possible to build a query language on top of the index/key system. However, since the index/key access is provided on top of a relational database, this would result in two layers of abstraction with the following downsides :

1. Highly increased complexity
2. Subpar performance
3. Reduced feature set for the database (for example transaction support will have to built from scratch, which is definitely no trivial task)

Additionally, it seems superfluous to implement a second layer of abstraction on top of a database to achieve functionality with the above-mentioned drawbacks, when this functionality is already present without the layers on top of it. That would be like going out to buy a red car, then have it professionally painted black, just to repaint it red again yourself.

Along with that, the implementational burden is not to be overlooked. Developing a full-featured SQL-like query language is no ordinary undertaking.

Beyond the features

there is another important aspect of these API's, namely the implementation in browsers.

Theoretically, we should go forward with only the Web SQL Database API, as it is clearly the superior specification for this project. Despite being the most feature-complete, we will demonstrate in chapter three why we need to support all of them.

OFFLINE WEB APPLICATIONS

The previous section has given us a solution to solve the performance problem of web applications. However, in order to enable users to continue interacting with web applications and documents, even when no internet connection is available, we need something more.
Because this appears to be a problem that has been faced by many web developers, facilities have been added to the HTML5 specification to allow offline web applications.

An overview of these facilities will now be discussed in this section. For a full, detailed overview of the specification we direct the reader to :
`http://www.w3.org/TR/offline-webapps/`

The application cache

The application cache of a web application can be defined as the set of resources needed for an application to work without an active internet connection.

An application cache consists of :

- Master entries
- Manifests
- Explicit entries
- Fallback entries

A *master entry* is indicated by the `manifest` attribute in the `<html>` opening tag of a HTML5 page. The attribute value has to be a valid URL referring to a manifest file.

Each page can have one *manifest* file, and all manifests of an application share the same origin.
The manifest (should) contain all resources necessary for a web application to be able to function offline. These resources are called *explicit entries*.

Furthermore an online whitelist (marked by the `NETWORK` keyword) can be added to define resources which should only be used online (and thus not be stored in the application cache); for example a remote database worker.

```
CACHE MANIFEST
FALLBACK:
 / /offline.html
NETWORK:
 *
```

The *fallback entries* are used when a file referenced in the manifest could not be found. An example of its use can be seen in this example: all resources have been put on the online whitelist, indicated by the asterisk in the `NETWORK` section. When the browser is offline and a page from this application is requested the browser will *not* display the cached version. Not even if the page is present in the "normal" browser cache, will it be displayed. The browser will, however, show the offline.html page.

Online/offline detection

A vital part of developing a web application with offline capabilities depends on efficiently and rapidly detecting whether the browser is online or offline, and hence, being able to respond in a proper way to this status.
Therefore the W3C and WHATWG have defined a browser attribute which allows one to instantly check whether the browser is online or offline [8]. The attribute `window.navigator.onLine` indicates the browser status. Whenever its value changes, an `offline` or `online` event is fired by the `window` object.

Although very useful, this feature is inherently unreliable; a browser in an online state connected to a network does not necessarily have working internet access. However, in the case of a lot of home users it is a good *indication* of the current connectivity status, and is therefore a useful addition.
Some browsers have built-in support for detecting when the operating system loses its network connection, e.g. Firefox on Windows and Linux [9]. Nonetheless extra steps have to be taken to be able to reliably detect the client's internet connectivity, e.g by pinging a reliable server at certain intervals.

DATABASE REPLICATION

Introduction

Database replication is the process where transactions performed on one database (the *source* -or *master* database) are propagated to one or more other databases (the *target* database(s)), to ensure consistency between all databases.
Multi-master reproduction allows data to be inserted and updated from all nodes, which is desirable for this application.

Data replication has traditionally been used to improve availability and performance of database-driven applications. Although not in a typically distributed manner, the application of database replication in this project is similar to the classical case.

To be able to understand everything discussed in this section, we need to introduce some *fundamental database concepts* first.

Fundamental database concepts

Database transactions

A database transaction comprises a unit of work performed on a database, and treated in a coherent and reliable way independent of other transactions. A database transaction, by definition, must be Atomic, Consistent, Isolated and Durable, which is often abbreviated as ACID.

- **Atomic -** Either all operations of the transaction are properly executed on the database, or none are.
- **Consistent -** The database must be in a consistent state before, during and after the transaction. If a transaction leaves the database in an inconsistent state, it is aborted and an error is reported.
- **Isolated -** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results cannot be visible outside of the transaction's scope.
- **Durable -** When a transaction reports it has been executed successfully, the changes to the database have to be persisted, even when there are system failures.

Serializability

In databases, when evaluating a DBMS, a transaction scheduling algorithm or anything related to transaction management, it is often demanded that they have the serializability property.
A transaction history is *serializable* if the resulting database state is equal to the outcome of its transactions executed sequentially without overlapping in time [10].

Serializability is considered the most important correctness criterium for concurrent transaction execution. It represents the highest level of isolation between transactions.

Serializability becomes even more important (and harder to implement properly) when dealing with distributed databases. The de-facto standard protocol for ensuring serializability in distributed databases is *strong strict two-phase locking* (SS2PL; also referred to as Rigorousness or Rigorous scheduling) [11]. In a nutshell, we could say that this protocol locks all data (preventing both reads and writes) related to a transaction until this transaction finishes (either by succeeding or aborting).

Obviously, this behavior is too strict for this project; at least not without some modifications. In the rest of this chapter we will try to find a replication technique that ensures serializability for distributed databases without using the SS2PL protocol.

Classification of database replication techniques

During the last decennia a plethora of algorithms have been developed and proposed for database replication. A comprehensive study of *all* ways to achieve reliable data replication is outside the scope of this thesis. Therefore, we have looked into most ways of classifying these techniques, after which a concise, well-informed choice can be made regarding the type of replication algorithm to be used.

Lazy vs. Eager

Synchronous, blocking (or **eager**) algorithms keep all nodes in sync at the same time. This is achieved by making the replication process part of the actual database transaction, and therefore preventing conflicts. For example, if a record is updated in two different nodes at the same time, one of the two will fail (i.e. based on a timestamp), thus preventing inconsistency.
One of the biggest disadvantages is that the risk of deadlock rises very fast with rising transaction size.

Asynchronous, non-blocking (or **lazy**) algorithms [12; 13] on the other hand, work independently on different replicas, and have to resort to conflict resolution during the synchronization phase. When a transaction is performed on a replica, the updates propagate asynchronously to other replicas.

Partial vs. Full replication

Full replication ensures that the full dataset is available at all nodes, at all times, while **partial replication** only provides a subset of the data at different nodes. The provided subset can differ from one node to another.
Data in a relational database can be vertically sub-setted, or horizontally. When viewing the database as a table, a vertical subset contains only some columns of the table. Horizontal sub-setting on the other hand only selects some rows (typically identified by the primary key column).

Optimistic vs. Pessimistic

Traditional replication techniques always try to preserve so-called *single-copy consistency*. This goal can be realized in many ways, but the base idea is always the same : access to data in a node is blocked unless this node is provably up-to-date. Hence, these techniques are called **pessimistic**. Many commercial systems (i.e. Oracle) implement this by electing a primary node to handle all accesses to a data object, thus guaranteeing this node is always up-to-date for that object.

An **optimistic** replication method [14] does not limit availability at a node in order to maintain consistency. Within each replica, a log is kept of the transactions that are optimistically allowed to execute, risking possible inconsistency with transactions running in other replicas. When the data is merged, these inconsistencies are detected and then resolved. In the worst case, transactions that have been committed must be later undone, then redone using a possibly different database state, and yielding possibly different results.

Lazy and optimistic (or eager and pessimistic, respectively) might seem two very similar properties to the reader, however there is an important difference : optimistic refers to the way data is allowed to be accessed, while lazy refers to how it is replicated.

Desired replication technique

For this project we want a :

- lazy,
- partial,
- optimistic

replication algorithm.

Lazy - Web applications have the downside that they typically no longer work when internet connectivity is lost. The goal of this project is to solve that problem by making use of a local database. If, however, we make the data replication part of the transaction, then we are back at square one.
Indeed, the database writes would fail (possibly even the reads, depending on how strictly eager the technique were implemented) because the replication would make the entire transaction fail. Additionally, the general consensus in the literature on this topic is that lazy replication is optimal for mobile and web applications [15-17].

Partial - Consider some of the typical characteristics of web applications. One of the obvious ones is that they almost always have different users, and strictly separated user-data. Firstly, it is superfluous to replicate the data of all users to every single user (unless in an attempt to create a distributed backup, which is far beyond the scope of this dissertation). Secondly, full replication could pose an enormous security risk, and possibly even violate the privacy of users. Lastly, for applications with a high number of users, this might require enormous bandwidth and storage requirements on the client side. When considering the average web users, both of these requirements are absent.

It should be noted that a lot of cases are imaginable where *some* data can or may be shared amongst different users. The burden of replicating the necessary data to allow this to work offline lies on the developer using the library. For example, a rule-based approach could allow for this.

Optimistic - From the moment a web application loses its connection to the remote server, it can no longer be sure about the consistency of the local database with respect to the remote database. If we used a pessimistic replication technique, no access to any data in the local database would be allowed once internet connectivity had been lost.
Taking into account the goals of this project, we need an optimistic algorithm by definition.

Update anywhere-anytime-anyhow replication

One simple replication technique that fits all the criteria listed in the previous section is the *update anywhere-anytime-anyhow* replication.

All nodes are considered equal, as are all data objects. When a node updates (called the root node for this transaction) its local data, a transaction is sent to every other node to apply the root's transaction updates.
This means it is possible for two nodes to update the same object at the same time, resulting in a conflict. To ensure serializability, the conflict is detected and the two concurrent updates are reconciled, so their updates are not lost.

Timestamps are typically used to detect and reconcile conflicts :
Each data object keeps a timestamp of when it was last successfully updated. When the root node propagates its updates it sends this timestamp along with the transaction. When this update is received at another node, it compares the timestamp with the one of the local replica.
If these are equal, the replica is updated. If they are not, the update is considered "unsafe", and the transaction is submitted for reconciliation. At this point, the reconciliation requires application-specific rules, or user interaction.

Scale-up pitfall

This technique works very well at low loads and with few nodes. However, when applied to a system with a high number of nodes, or when nodes are offline for a long time, it behaves quite differently. This is called a scale-up pitfall.

Because of the higher transaction rates in larger scale systems, suddenly the reconciliation rate grows very fast. The outcome is that data updates conflict more and more, resulting in the nodes diverging further and further away from each other. Finally, every node rejects updates from all other nodes. The database is inconsistent, with no apparent way of restoring it to a consistent state without data loss.

Gray et al. have proven that a ten-fold increase in nodes increases the reconciliation rate by a factor of a thousand [15]. Furthermore, they have also shown that the reconciliation rate quadruples when the average time the nodes are offline doubles.

This is an unacceptable behavior, especially when factoring in that web applications are easily used by thousands of users (nodes). On top of that the mobile internet is at the rise now, resulting in users that are offline a lot of the time.

Two-tier replication

A great improvement over the basic update anywhere-anytime-anyhow technique is the two-tier replication technique, suggested by Gray et al. [15]. It does not suffer from a scale-up pitfall and greatly reduces the number of needed reconciliations.

Firstly, we need to assign an owner (called the **maste**r) to each data object. Updates are first applied by the owner, and then propagated to other nodes. Each data object can have a different master.

All nodes are no longer equal, we distinguish two kinds of nodes :

- **Mobile nodes** are usually offline, have a replica of the database and may originate tentative transactions. Some data objects can be mastered by a mobile node.
- **Base nodes** are always connected and store a replica of the database. They master most data.

Data objects in a mobile node might co-exist in two versions :

- The **master version** is the most recent version received from the object's master.
- The **tentative version** holds the most recent values due to local updates.

Analogously, there are also two kinds of transactions :

- A **base transaction** works only on master data, and produces new master data.
- A **tentative transaction** works on local tentative data. A tentative transaction produces a new tentative version and a base transaction to be run at a later time on the base node.

To prevent the need for numerous reconciliations, an **acceptance criterion** is defined for all types of data objects : a test on the output data of the transaction. For example, a criterium on a bank balance could be that it is not allowed to be negative.
Whenever a base transaction, generated by a tentative transaction, is run, the result set is verified by its acceptance criterion. Thus, base transactions generated by a tentative transaction may fail, or even produce different results.

Mobile node connects to a base node (tier 1)

The following steps are performed when an offline node goes online and connects to a base node :

1. Send all replica updates for data objects mastered at mobile node.
2. Send all tentative transactions in-order.
3. Discard all tentative object versions, as they will soon be refreshed by the base node.
4. Accept replica updates from base node.
5. Accept notification of the success or failure of each tentative transaction.

Base node is contacted by a mobile node (tier 2)

The following steps are performed when a base node gets a synchronization request from a mobile node :

1. Send all replica updates.
2. Accept replica updates for data objects mastered at the mobile node.
3. Accept the in-order list of tentative transactions.
4. Reprocess every transaction in the order they were ran on the mobile node, taking into account the acceptance criteria. If a transaction fails its acceptance criterion, the base transaction is aborted, and a notice is returned to the mobile node.
5. Propagate all affected data objects to all other replica nodes.

When all tentative transactions have been reprocessed as base transactions, the mobile node is consistent with the base node.

Conclusion

Although not perfect, this technique allows one to replicate data without the risk of rendering the database inconsistent and unusable. The base database is always in a consistent state, while clients are allowed to make tentative updates. Furthermore, replicas at mobile nodes always converge to the base state and cannot diverge to an unusable state.

When implemented strictly (as described above), no reconciliations are needed. However, when the base node processes the mobile node's transactions, different results may be produced at the base node. While this is acceptable for most applications, it is definitely not for all. Moreover, the required acceptance criterions are very application-specific, and could - in some cases - be replaced by conciliation to fit the application. For a very data-oriented application, this will prove to be a tedious and elaborate work. On top of that, it is difficult to maintain as all criteria have to be rechecked when changing or updating the database structure.

Transaction-level result-set propagation replication

Transaction-Level Result-Set Propagation, or TLRSP, is a replication technique developed to fit to modern, mobile client database systems. It was first introduced by Ding Z. et al. [18] in 2001 and has been shown to perform better than the two-tier replication technique. It requires less data transfer between nodes and also needs less extra data to be saved locally (at the client) for synchronization properties. Furthermore it requires no extra effort (i.e. acceptance criteria in the two-tier replication) from a developer's viewpoint.
As in the two-tier replication scheme, we distinguish mobile nodes and base nodes. Base nodes are considered to be stable and always connected, while mobile nodes are mostly disconnected and go online once in a while.

A mobile node can be in one of three states :

1. **Consistent state -** The mobile node's database has been updated recently by a base node and is consistent with the base node's dataset.
2. **Accumulating state -** The mobile node is offline, and accumulating transactions. These transactions are performed on the local database replica, while information about the transactions is saved. Instead of logging every transaction, a simplified log structure is used : the Read, Write and ResultSet (to be defined below) of the transactions are stored.
3. **Resolving state -** When the mobile node reconnects to a base node, it is in the resolving state. Firstly, the locally committed transactions are submitted to the base node for conflict detection. Then the base node performs its reconciliation techniques. Finally, the updated objects at the base node are sent back to the mobile node to refresh the local data. The mobile node is in the consistent state again, and all logs are cleared.

Reconciliation strategies

To explain the rather complicated reconciliation strategies employed by the TLRSP replication technique, we need to define some terms first :

- **UTQ** is the Upload Transaction Queue and contains all logged information for transactions accumulated at the mobile node while it was offline. The transaction information is ordered in the same way the transactions were performed at the mobile node.

In the following definitions and theorems we fix a UTQ consisting of the transactions $T_1, T_2, \dots T_n$ and use the notation $(T_1 \ldots T_i)$ to refer to the processed transactions up until T_i in the UTQ.

- The **ReadSet(T_i)** of a transaction T_i contains all the data objects read by the transaction.
- The **WriteSet(T_i)** of a transaction T_i contains all the data objects edited by the transaction.
- The **ResultSet(T_i)** of a transaction T_i contains all the pairs of data items in the WriteSet(T_i) and their new value under transaction T_i.
- The **AccessSet(T_i)** of a transaction T_i is the union of the ReadSet(T) and WriteSet(T), for every transaction T in $(T_1 \ldots T_i)$.
- The **ReadableSet(T_i)** of a transaction T_i contains all data objects of AccessSet(T_i) that have the same value at the start of the transaction at the mobile node, as the value at the base node when verifying the transaction.

We can now state the following theorems :

Theorem 1 - For any transaction T_i : if *ReadSet(T_i)* is a subset of *ReadableSet(T_i)*, then T_i can be committed globally at the base node. That is, *ResultSet(T_i)* can be written in the base database.

Theorem 2 - For every transaction T_i : *ReadableSet(T_i)* is a subset of *AccessSet(T_i)* and consists of data objects that have not been updated by the base node since the last synchronization between the mobile and base node.

Theorem 3 - If the *ReadSet(T_{i-1})* is a subset of *ReadableSet(T_{i-1})*, then *ReadableSet(T_i)* is the union of *ReadableSet(T_{i-1})* and *WriteSet(T_{i-1})*. Otherwise *ReadableSet(T_i)* is the *ReadableSet(T_{i-1})* minus the *WriteSet(T_{i-1})*.

We have included the theorems here solely to support the algorithm defined in the next section. For an elaborate proof of these theorems, we refer the reader to [18].

ReadableSet(T_i) of every transaction can be determined iteratively using theorem 2 and 3. We can then validate every transaction using theorem 1.
After validation, all transactions in UTQ are divided into two sets : **CancelSet** and **CommitSet**.

The **CancelSet** contains all WriteSets of transactions that have been deemed **invalid** in the reconciliation process and is sent to the mobile node. The mobile node can then cancel those writes to be in sync with the base node.
The **CommitSet** contains all WriteSets of transactions that have been deemed **valid** in the reconciliation process and are committed at the base node.

Conflict detection and reconciliation algorithm

Using the above theorems, we can define the following algorithm for conflict detection and reconciliation at the base node. This algorithm will be executed the mobile node enters the resolving state, i.e. going back online after accumulating data in an offline state.

Input : UTQ = (T_1, T_2, … T_n)
AccessSet(T_n)

Output : CommitSet
CancelSet

```
1. Initialize CommitSet and CancelSet as the empty set
2. Shared-Lock[9] all data objects in AccessSet($T_n$)
3. Initialize RS($T_i$) to ReadableSet($T_i$) for every transaction $T_i \in UTQ$
4. Process every transaction $T_i \in UTQ$ in-order :
     a. Exclusive-Lock[10] all objects in WriteSet($T_i$)
     b. If ReadSet($T_i$) is a subset of RS($T_i$)
          i.  Add WriteSet($T_i$) to CommitSet
          ii. Add WriteSet($T_i$) to RS($T_{i+1}$)
     c. Else
          i.  Add WriteSet($T_i$) to CancelSet
          ii. Subtract WriteSet($T_i$) from RS($T_{i+1}$)
5. Remove all shared locks
6. Commit everything in CommitSet
7. Remove all exclusive locks
8. Send CancelSet to base node
```

Conclusion

We have started off with the most basic of replication techniques that fitted this project's requirements. After identifying its flaws, we discussed the two-tier replication technique. Although this "tried-and-tested" technique (and a multitude of variations) has been implemented by countless DBMS's, it did not prove to be a perfect fit for this project. This brought us to the Transaction-Level Result-Set Propagation technique, which appeared to be perfect. We have therefore chosen this replication technique as the one that will be used in this project.

[9] A shared lock allows other processes to read the object, but not write to it.

[10] An exclusive lock block all reads and writes from other processes.

Chapter 3
State of the art

Browser Rendering Engines

Since the HTML 5 standard is currently in the "Last Request for Comments" state, adaptations in browsers are - at the time of writing - quite immature. On top of that, a lot of browser developers keep their plans tightly under wraps, which makes it difficult to get an overview of the currently implemented and planned features.

Therefore, we have developed a web application to test the implemented features in browsers. The application uses only W3C compliant HTML 5 (compliant to the working draft) and JavaScript.

The following features were tested :

- Web SQL Database
- Indexed Database
- Application Cache
- Local Storage[11]
- Online/Offline detection

WebKit, Mozilla Gecko, Opera Presto and Microsoft Trident have been tested. Together these cover more than 98% of the current browser market on all popular operating systems. Besides the test results, we have also included all official statements referring to any of the storage APIs by the browser vendors.

WebKit test results

WebKit is an open source web browsing engine that began as a branch of KHTML and KJS. It is best known for its implementation by Apple in Mac OS X' default browser, Safari. Currently it is being developed by the KDE project, Apple, Nokia, Google and others. Furthermore it is being used in a plethora of different browsers and platforms : Apple iPhone MobileSafari & OS, Google Chrome & Chromium, Symbian OS, Valve Steam gaming platform, Palm WebOS, …

Official support

WebKit officially supports Web SQL Databases since versions 525 [19]. On top of that they provide a tool called the Web Inspector to, among a lot of other things, browse the local database.
Furthermore there are a few known bugs and some slight differences from the specification.

[11] Local Storage (also referred to as DOM Storage) can be viewed simplistically as an improvement on cookies, providing much greater storage capacity for key/value pairs (typically up to 5 MB). It has been included here for completeness.

Google has announced that they will be adding Indexed Database support to the Chromium project [20], essentially bringing the Indexed Database API to WebKit. The first implementations have been recently started [21].

Test results

Browser	OS	Web SQL DB	Indexed DB	App Cache	Local Storage	On/Off events
Safari 4.0	Mac OS X 10.5, 10.6	✓		✓	✓	✓
	Windows XP, Vista, 7	✓		✓	✓	✓
Google Chrome 4.0	Mac OS X 10.5, 10.6	✓			✓	✓
	Windows XP, Vista, 7	✓			✓	✓
Google Chrome 5.0	Mac OS X 10.5, 10.6	✓		✓	✓	✓
	Windows XP, Vista, 7	✓		✓	✓	✓
	Ubuntu 8.10	✓		✓	✓	✓

While testing we encountered two bugs :

- The specification defines a callback function to be called when the local database was successfully opened. However, using Safari this callback was never called, even though the database was successfully opened. In Google Chrome this bug was not present. After some research we found out this is due to a bug in WebKit[12], which has been recently fixed. Chrome, unlike Safari, uses a more recent WebKit version, which includes this bug fix.
- Another (undocumented) bug was found using both Google Chrome and Safari. This bug relates to a wrong error code being returned when a transaction fails and has been submitted to the WebKit bug tracker[13].

Browser market share

At the time of writing, WebKit-based browsers comprise about 13% of the browser market[14].

Mozilla Gecko test results

Gecko is a browser layout engine currently being developed by the Mozilla Foundation, and serves as the basis of the popular Firefox browser. It is a very standards-compliant, open source engine written in C++ that runs on various operating systems : Windows, Linux, BSD, Mac OS X, Solaris, …

[12] Bug ID : 34726 | Fixed in build r55834 on 2010-03-11

[13] Bug ID : 36513 | Submitted on 2010-03-23

[14] According to W3Counter (http://www.w3counter.com)

Although most famous for its use in Firefox, Gecko is also used in different browsers and programs such as Mozilla Thunderbird, Camino, SeaMonkey, …

Official support

Mozilla has an official roadmap for the Gecko 1.9 branch [22], which is the current one. The Indexed Database API has been added to the 1.9.3 roadmap, which is expected to make its first appearance in Firefox 4.
According to the development status pages, work on the implementation has started and is actively being continued [23].
Mozilla's Director of Firefox has indicated in a recent product plan presentation for Firefox 4 that no plans to support the Web SQL Database API exist [24].

Test results

Browser	OS	Web SQL DB	Indexed DB	App Cache	Local Storage	On/Off events
Firefox 2.0	Mac OS X 10.5, 10.6					✓
	Windows XP, Vista					✓
	Ubuntu 8.10					✓
Firefox 3.0	Mac OS X 10.5, 10.6			✓		✓
	Windows XP, Vista			✓		✓
	Ubuntu 8.10			✓		✓
Firefox 3.5	Mac OS X 10.5, 10.6			✓	✓	✓
	Windows XP, Vista			✓	✓	✓
	Ubuntu 8.10			✓	✓	✓
Firefox 3.6	Mac OS X 10.5, 10.6			✓	✓	✓
	Windows XP, Vista, 7			✓	✓	✓
	Ubuntu 8.10			✓	✓	✓
Firefox 4 alpha 5	Mac OS X 10.5, 10.6		✓*	✓	✓	✓
	Windows XP, Vista, 7			✓	✓	✓
	Ubuntu 8.10			✓	✓	✓
Camino 2.0	Mac OS X 10.5, 10.6			✓		✓

* **Important note -** On Mac OS X, a Mozilla Developers Preview build reports support for the Indexed Database API. The implementation does not appear to be working at the time of writing though, which is logical considering the current development progress of this feature.

Browser market share

At the time of writing, Firefox comprises about 33% of the browser market.

Opera Presto test results

Presto is the layout engine used in Opera, and has been developed by Opera Software. Presto is available only as a part of the Opera browser or related products. The source or binary forms of the engine are not publicly available. Presto is a very efficient engine and at the time of writing the fastest available web rendering engine [25].

Official support

According to Opera Software's version history [26] and supported web standards list [27], Web SQL Databases have been introduced in Presto 2.4 with the 10.50 pre-alpha release of Opera. Final support and full implementation of the standard has been realized in Opera 10.50 [26].

Test results

Browser	OS	Web SQL DB	Indexed DB	App Cache	Local Storage	On/Off events
Opera 9.6	Mac OS X 10.5, 10.6					✓
	Windows XP, Vista, 7					✓
	Ubuntu 8.10					✓
Opera 10.5	Mac OS X 10.5, 10.6	✓			✓	✓
	Windows XP, Vista, 7	✓			✓	✓
	Ubuntu 8.10	✓			✓	✓

Market browser share

Despite being a very standards-compliant, feature-rich and fast browser, Opera has only been able to achieve a 2.1% market penetration.

Microsoft Trident test results

The most used layout engine throughout history, Microsoft Trident, is best known for its use in Microsoft Internet Explorer. Although some odd versions of Internet Explorer use a different engine (i.e. Microsoft Internet Explorer 5 for Mac), every popular version of Internet Explorer has used this engine.
Introduced in October 1997 in Internet Explorer 4, it is still in use today and has been steadily updated for each Internet Explorer iteration. Internet Explorer 8 (the most current, stable version) uses Trident 4.0, and was the first Microsoft browser to pass the Acid2 test[15]. Version 5 will be used in Internet Explorer 9, and will - according to Microsoft - improve compliance with web standards and add support for new web-technologies.

Official support

Microsoft publicly endorsed the Indexed Database specification on its Internet Explorer blog [28] : "Together with Mozilla, we are excited about a new design for local storage called Indexed DB. We think this is a great solution for the Web," said program manager Adrian Bateman.
On more than one occasion the Indexed DB specification has been praised by the main Software Architect in the Data and Modeling Group at Microsoft, Pablo Castro [29; 30]

Test results

Browser	OS	Web SQL DB	Indexed DB	App Cache	Local Storage	On/Off events
IE 6	Windows XP					✓
IE 7	Windows XP, Vista					✓
IE 8	Windows XP, Vista, 7				✓	✓
IE 9 tech preview	Windows 7				✓	✓

Market browser share

Microsoft Internet Explorer holds an approximated 51% share of the browser market.

[15] Acid2 is a test page published and promoted by the Web Standards Project to expose web page rendering flaws in web browsers and other applications that render HTML. More information can be found at http://www.webstandards.org/action/acid2/

Conclusion

A stable implementation of the Web SQL Database specification has only been realized in Webkit-based browsers (i.e. Google Chrome & Apple Safari) and the most recent version of Opera. Together these browsers comprise no more than 15% of today's browser market.

No working implementation of the Indexed Database specification could be found at the time of writing. Nonetheless, this will prove to be a very important storage API in the near future, as most major browser vendors (i.e. Microsoft, Mozilla and Google) have announced their appraisal for it.
Microsoft, Mozilla and Google cover more than 90% of the browser market with their respective browsers; hence their decision to support *only* the Indexed Database specification (safe for Google) might tip the scale in favor of the Indexed Database specification.

It is too early to say which of the two local database specifications will come out on top. On the one hand, Web SQL Database has the advantage of already working implementations in fairly popular browsers. On the other hand, Indexed Database has the support of the biggest browser vendors, with implementations from all of them on their way.

Consequences for this project
As it seems impossible to decide at this point in time whether to support Web SQL Database or Indexed Database in our project, we are left with only one possibility : the local database access of this project will have to be modular, easily up-dateable and allow different backends to be interchangeable.
In the next chapters we will establish how this will be achieved.

EXISTING SOLUTIONS

In this section we will look into existing software that someway or somehow tries to fill the same gap as this project, or that has a similar goal.

Firstly, JavaScript implementations for local data storage will be described, followed by an overview of existing middleware for database replication.

Javascript implementations

Persistence.js

Persistence.js is a simple asynchronous JavaScript *object-relational mapper*[16] (ORM) library. It makes use of the HTML 5 Web SQL Database, while it also supports Google Gears' local SQLite database.

Its main target is to allow developers to create offline-capable web applications. However no synchronization with remote databases seems to be on the roadmap.
It has no dependencies on any other frameworks, other than the Google Gears initialization script.

Developed by :	Zef Hemel
License :	MIT License
More info :	`http://github.com/zefhemel/persistencejs`

ActiveRecord.js

ActiveRecord is a stand-alone object relational mapper which can operate using an in memory hash table, or with an SQL back end on the Jaxer platform (SQLite and MySQL), Adobe's AIR (SQLite) or Google Gears' local database. Support for the HTML5 Web SQL Database specification is planned.
Contrary to what we are trying to achieve in this project, ActiveRecord is **synchronous**.

They also promote a "synchronization" feature on their website, which is not the same as the synchronization/replication described in this dissertation. Synchronization in the context of ActiveRecord means that a result set can be kept in sync with the database. If a record belongs to the result set of a certain query, and this record is updated elsewhere, the result set will also be updated, thus preventing having to query the database again.
For example, imagine we have a database of users. If we query the database for all users named "Bob", we get a result set containing all Bobs. If Bob now logs in, and changes his e-mail address, then both the record in the database and the record in the result set will be updated.

Developed by :	Aptana Inc. / ActiveJS
License :	Completely free (`http://github.com/aptana/activejs/blob/master/LICENSE`)
More info :	`http://activerecordjs.org/index.html`

[16] For more information on this subject we refer the reader to the glossary

Web storage portability layer (WSPL)

The web storage portability layer (WSPL) is a JavaScript library for providing a single portable asynchronous API to access a database provided by either Google Gears or HTML5 Web SQL Database.

WSPL is a thin abstraction layer on top of the query language for SQLite (an SQL variant), to hide the underlying local database technology being used from the developer.

Developed by :	Rob Kroeger
License :	Apache License 2.0
More info :	http://code.google.com/p/webstorageportabilitylayer/

Database replication middleware

Although a lot of middleware has been developed through the years to achieve database replication, only a few are mentioned here. We have chosen to only list solutions capable of data replication between a remote web database, and an SQLite (or similar) database.

It should also be noted that all major DBMS's, e.g. Oracle, MySQL, PostgreSQL, etc. have built-in support for replication. However, this support is almost always restricted to databases running on the same DBMS, and therefore not relevant for this project.

Pervasync

Pervasync is a data synchronization system for database systems such as Oracle, MySQL and SQLite. It enables users to synchronize multiple local databases with a central database without writing a single line of code.
Even SQLite databases embedded in browsers by the Google Gears plugin are supported as a client.

To install Pervasync server, a Java servlet container, e.g. Apache Tomcat, is required. The sync server includes a standard J2EE servlet application, which needs to be deployed on an application server or a J2EE container.
The downside is that it is impossible to use in a shared hosting environment. Many developers of small web applications do not have a full featured J2EE application at their disposal.

The price for a single server Pervasync license is $999. On top of the initial server license, a fee has to be paid per client. This replication solution is aimed at organizations who wish to replicate certain data to their employees or customers. The difference with a web application is that in the former case the organization is able to control who uses the replication solution (and thus can predict the costs). In the latter however, a developer cannot (in most cases) control who uses his web application. He is therefore also unable to predict the incurred costs, which is unacceptable for this project.

More info : http://www.pervasync.com

Microsoft Sync Framework

Microsoft describes its Sync Framework as a comprehensive synchronization platform enabling collaboration and offline operation for applications, services and devices. Developers can build synchronization ecosystems that integrate any application, any data from any store using any protocol over any network. Sync Framework features technologies and tools that enable roaming, sharing, and taking data offline.

It is a mature, very comprehensive framework that allows both partial and full replication, different conflict resolution strategies, different synchronization strategies and virtually any type of database or data provider.

To get started with the Microsoft Sync Framework one needs to install the Software Development Kit (SDK) which is only available for Windows and requires the .NET-framework. Furthermore, developing applications using this framework is only supported using the Visual Studio development environment.
To be able to operate efficiently, the Microsoft Sync Framework requires extra server components to be running. This severely limits the applicability to existing web applications, and does not make it a viable option for smaller web developers. Combined with the closed nature of this framework, and the operating system restrictions we believe this not to be a good solution for this project.

More info : `http://msdn.microsoft.com/sync`

Conclusion

JavaScript implementations

It appears some implementations have been built on top of the Web SQL Database specification, however most of them are object-relational mapper (ORM) libraries or abstraction layers.

ORM libraries hide the raw database accesses, by allowing the developer to create native objects, which are automatically persisted by the library. The developer only needs to define his objects, and the relations amongst them, while the ORM library takes care of the table creation, structure, queries, etc. It allows a developer without any specific database knowledge to develop database-driven applications.
For this project, we do not need ORM functionality, because this typically does not allow us to define the way the physical database tables are structured. If we want the information in that database to be accessed both by a local JavaScript interface and a remote server-side interface, we need to be able to define the table structure. If not, we would have to re-implement the ORM logic at the server side to be able to use the same table structure, which would defeat the purpose of an ORM library in the first place.

The *database abstraction layer*, on the other hand, is very relevant for us. We do not want developers to be restricted to a certain local database backend (as indicated in the conclusions of the former section on web browsers), so we will need a database abstraction layer.

However, the existing abstraction layers are limited to the Web SQL Database API and the Google Gears Database API. We want our solution to support more backends, while leaving room for future, new backends.

Database replication middleware

Although only two solutions have been listed in the section on database replication middleware, they both represent a general class of replication middleware. More similar solutions could be listed, however they would result in the same conclusions.

It appears there exists no open, light-weight database replication middleware, which would allow us to replicate data from a generic client-side database to a server-side web application database without needing extra server components.
Although obviously some server-side logic is necessary we believe this can be achieved by making use of existing web application technologies, without having to resort to installing extra server daemons or extensions. How we accomplish this will be explained in the next chapters.

Chapter 4
Project Vision

Motivation

We started off this project with the goal of enabling near desktop performance for web applications. Along the way we added a second main goal, namely allowing offline usage of web applications.

To accomplish these two main goals, there are two requisites :

- A way of saving, and accessing data locally - We have introduced the relevant techniques to achieve this and decided which ones will be used.
- A way of synchronizing this locally saved data with remote data - After classifying our problem we were able to determine the best technique to perform database synchronization.

In the state of the art we established that no solution exists today that unites heterogenous database replication with local browser databases and remote web databases.

Mission Statement

Develop a JavaScript library to allow developers to create database-driven web applications with desktop-like performance by using a local browser database, including support for transparent offline operation and automatic synchronization between local and remote database(s).

Project name

The project name will be **JuSy**.

JavaScript is often abbreviated as JS by web developers. Since the main part of this project is written in JavaScript we wanted to this to be reflected in the name.
The "Sy" part is short for synchronization, which is the main feature of the project.

Key Features

Automatic synchronization

The most important feature of this project will be the automatic, and completely transparent, synchronization of (a subset of the) remote database with the local browser database.
Upon first use of a web application using this library, the initial data will be replicated from the remote database to the local browser database. Following this, all transactions on either the remote or local database will be replicated as soon as possible. This is particularly interesting for mobile applications, as network connectivity can be interrupted arbitrarily.

Detection of internet connectivity

Applications using JuSy will be able to realize instantly when internet connectivity has been lost, and respond appropriately. Data accesses will be automatically switched to the local database, and writes will be saved, so that when internet connectivity is restored, the remote database can be synchronized with the local one.

Partial or full data replication

Developers will we able to choose whether they want a complete database/table replicated, or only records with a specific distinction. A very interesting example of this would be to only replicate records for the logged in user.

Cross-browser

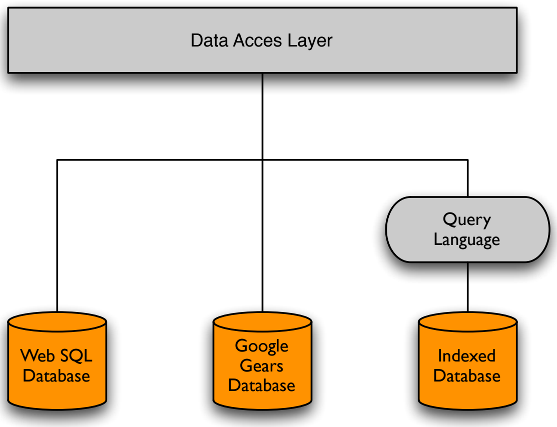
The JuSy library will not depend on any specific browser or operating system. It is our goal to support at least one local storage backend for every popular browser on every popular platform.

Main Quality Attributes

Modifiability

As we have demonstrated in chapter 2, more than one solution has been proposed to save data client-side. In chapter 2 we concluded that it is impossible to decide at this time which one will come out on top. Therefore we need to support multiple local storage backends. Developers can implement a local storage backend, which has to be pluggable in the JuSy library without any modifications to the library itself.

Furthermore, the specific backends have to be easily up-dateable. At the time of writing, the Web SQL Database specification has reached an impasse: all implementors have used the same SQL backend (SQLite). However, to proceed along a standardization path there should be multiple independent implementations. This means that in the future, small differences between implementations of this specification may arise throughout different browsers. Our library has to be able to adapt to this easily and rapidly.



Usability

The implementation should be completely transparent for the end user, i.e. he or she should not notice when the application switches to its local database because of loss of internet connectivity. Furthermore, the synchronization process between the local and remote database should be transparent for developers using the library.

Another aspect of usability lies in being able to update existing client-side databases. Typically, web application are often updated, which also means the database has to be updated, and possibly needs structural changes. The library should be able to detect which database scheme the client is running, and update it accordingly, without losing data.

Performance

Applications using JuSy will have instant database access, while adding or editing of data should also be propagated instantly through the application. One exception will be the "cold start", when the application has to synchronize for the first time with the remote database server. Once this has been taken care of, instant data manipulation can be achieved by performing all operations on the local data-sources while asynchronously propagating these to the remote server.

Testability

JavaScript has been deemed a sub-standard scripting language, because of the fact it is quite easy to get started with, and a lot of bad code is floating around the web. Nonetheless JavaScript is a very powerful, dynamic, object-oriented general-purpose programming language.
We plan on developing a unit-tested library, complying with today's typical software design standards.

Chapter 5
Scenarios

USER SCENARIOS

A user runs an application for the first time

Main success scenario

1. The user creates a new account, supplying login credentials.
2. The credentials and user info are verified, to ensure a legit user signup.
3. It is determined by the library which options are available to store data locally, and an intelligent choice is made.
4. Depending on the decision made in step 3, the local database is created and the necessary queries are run to build the tables and structure.
5. The user accesses the application, using the remote database, while in the background, the databases are synchronized asynchronously.
6. When the synchronization has been finished the user is informed that offline operation is available.

Extensions

1. The user already has a user account for the web application, but logs in from a new location/computer for the first time.
1. The user indicates that he is logging in from a public/shared/… computer, and thus does not want any data to be copied to this system. No databases are setup, and no data is replicated to the system.
3. No suitable option for local data-storage can be determined. The user is informed about the lack of offline capabilities and is instructed to upgrade to a more modern browser.
5. The user exits the application before the replication phase has been finished. He or she is cautioned that offline operation will not be available, and is proposed the option to leave the application logged out, while synchronization continues. He/she is requested not to close the browser window.

Triggers

- A new user signs in for a web application using JuSy.
- An existing user logs in to this application from a different computer/location than usual.

Pre-conditions

- The user uses a modern, standards-compliant browser, i.e. Firefox 3.5+, Google Chrome 4+
- The library has been properly integrated with the web application, rules have been setup as to which data has to be replicated.
- The necessary steps have been taken to allow the client-side database to synchronize with the remote one.

Post-conditions

- A client-side database has been set up on the user's computer.
- The necessary data has been replicated to the local database to ensure that offline operation is now possible.

Conclusion

This scenario displays the typical usage of a web application, with or without offline capabilities. However, what is illustrated here is that adding these capabilities does not have a big, negative impact on the initial experience. Indeed, the extra effort to allow offline operation for an end-user is minimal.
The only exception takes place when the user is instructed to update his/her browser. While this may be construed as annoying by the end-user, it is good practice to inform the user about the possible security threats, performance issues, etc. of using an old browser. Furthermore, this introduces an extra stakeholder in the value chain, namely browser builders. For example, currently Canonical has a very lucrative deal with Yahoo, to suggest Yahoo Search as the default search engine in Ubuntu Linux. Analogously, deals could be made with browser builders as to which browser to suggest the user to upgrade to.

The minimal extra effort is by far outweighed by the extra value added by being able to use a web application without internet connection.

Internet connectivity is lost while using a web application

Main success scenario

1. The user has logged in to the application and is using the application in a typical way.
2. Internet connectivity is lost.
3. The application detects the loss of connectivity within a reasonable amount of time, and notifies the user in-obtrusively (for example, an icon change, or a notification in a status bar).

4. The application switches internally to "offline" mode, all data is saved locally, and all necessary actions are taken to ensure that a swift re-synchronization with the remote database can occur as soon as connectivity is restored.
5. The user continues using the application in a normal way. If a resource is requested which is not available offline, the user is notified about this. Optionally, if the developer has chosen to implement this, the user is given the option to make sure this resource is available offline in the future.
6. Internet connectivity is restored, all local data is replicated to the remote database in the background, normal online operation resumes. The user is notified in a subtle way, e.g. by a non-modal dialog.

Extensions

1. The user has not logged in yet, but wants to use the application without internet connectivity.
2. Alternatively, the user could switch to offline mode him/herself. A reason for this could be to avoid network congestion, to save on bandwidth usage, …
3. If this is the first time the user loses connectivity while using this application, the developers could have chosen to inform the user more intrusively about the limitations and ramifications of working offline. The user can then choose not to display this message in the future, after which a more subtle offline notification will be used again.
5. Resources unavailable in offline mode could be immediately marked when entering offline mode, e.g by greying them out in the menu, adding an unavailable icon, …

Triggers

- Internet connectivity is lost while using a web application.
- No internet connection is available when logging in to a web application.
- The user choses to work in offline mode.

Pre-conditions

- The user has used this application at least once on this computer, and he or she has allowed the setup and initial synchronization to finish.
- The user is using the same browser that he or she usually uses for this application. Furthermore the browser supports at least one of the local data-sources mentioned in the state-of-the art.
- The used browser implements the Offline Web Applications specification, allowing efficient and fast detection of browser connectivity.

Post-conditions

- The added (edited, deleted) data during the offline usage has been added to the remote databases, both local and remote data is back in sync.

Conclusion

This scenario illustrates the ease of use, and seamless user experience which can be achieved by enabling offline operation for a web application.

Besides the obvious advantage of being able to continue working without an active internet connection, one can imagine more uses :

- Imagine a user who has to pay for his internet connection on a per-connection basis. Using the application offline for a long time, and only enabling internet at the end of the day for the synchronization, could imply a great cost reduction for this user.
- A company with limited bandwidth could instruct its employees to use their web application in offline mode during day, and enable online mode at the end of the day. This way, scarce bandwidth can be saved during the day for other, possibly more important business processes, while the web application data can be synchronized when this bandwidth is not needed.

A user logs in to a web application with an updated database structure

Main success scenario

1. The user logs in to the application
2. JuSy detects the local database version
3. Every migration to update the user from his current database version to the most recent version is executed
4. The local database is resynchronized with the remote database

Extensions

4. No resynchronization is necessary

Triggers

- The developer has updated the database structure of the web application, some time after the user logged in for the last time.

Pre-conditions

- The user has used this application at least once on this computer, and he or she has allowed the setup and initial synchronization to finish.
- The user is using the same browser that he or she usually uses for this application. Furthermore the browser supports at least one of the local data-sources mentioned in the state-of-the art.
- The developer has included all migration steps in the application to allow migration from any version to the most recent version.

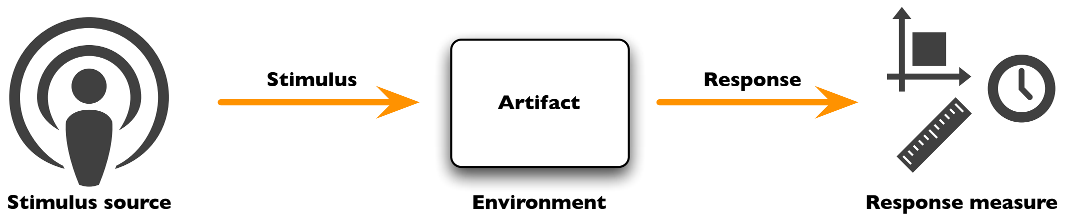
Post-conditions

- The local browser database has been updated to the most recent version, and is synchronized with the remote database.

Conclusion

This scenario illustrates how developers can easily update their database structure, without having to worry about the structure of the local browser database. It show how JuSy prevents reading (writing) to a database with an out-of-date structure and hereby preventing unexpected behavior, or possibly even data corruption.

QUALITY ATTRIBUTE SCENARIOS



Performance

The idea for this project started out with the question : "Can we build a database-driven web application that performs like a native application?". Hence, very strict measures will be defined in this section to ensure the performance demands are met.

Stimulus

A user logs in a JuSy application for the first time.

Response

The necessary database(s) and tables are created locally.
Synchronization between remote and local database is initiated.

Response measure

Local databases should be created instantly, as part of the login process.
The data synchronization is done asynchronously, while the user uses the application in a normal way. Normal (remote) operation of the web application should not be slowed down, or altered in any way by the synchronization process. However, second to the priority of the normal application use, the synchronization process should be completed as fast as possible.

Stimulus

A user is using a JuSy application with his usual browser, on his own computer, where the initial setup and synchronization has been fully completed.

Artifact

An intensive database-driven operation is executed.

Response

The necessary data is created/removed/updated/deleted (CRUD) on/from the local database first. Subsequently, the database is synchronized in the background with the remote database.

Response measure

The implementation of interacting with the local database has to be highly efficient, and no noticeable delay can exist. Furthermore, the synchronization background process should not interfere or slow down any other remote requests, if they exist.

Stimulus

A user updates a shared data object of a web application, used concurrently by different users, while the users is connected to the internet

Artifact

A shared data object.

Response

The update is propagated to the remote database.

Response measure

When a user updates a data object the object is first updated in the local database. As soon as this has finished, the update has to propagate to the remote database. There should be no extra delay besides the normal transmission and processing delays.

Modifiability

With the current state of the web browser world, and the fact that none of the specifications mentioned in the state of the art chapter for storing data client-side have reached a final state, modifiability has to be an important quality attribute.
One might even say it would be foolish to not decouple the local storage part of this project from the rest of the functionality.

Stimulus

A developer wants to add a new local storage technology to JuSy.

Artifact

Local storage component.

Response

Implementation of the new technology is added to the data-storage component.

Response measure

The time needed to implement should be minimal, and completely independent of the rest of the library. Furthermore, the new technology has to be pluggable in the existing library, without any alteration to JuSy.

Stimulus

A developer wants to use only the local storage or data synchronization component, independently of JuSy.

Artifacts

Local storage component.
Data synchronization component

Response

The components are available separately.

Response measure

Both components should be very loosely coupled, as should the rest of the library. The main components, however, should be completely independent of each other and easily interchangeable

Usability

The most important part of a software project is how the *user* experiences it. This is defined in part by the performance, but the rest will be defined here.

Stimulus

A user loses internet connectivity while using a JuSy application.

Artifact

Complete JuSy library

Response

JuSy detects the loss of connectivity, and reports this unobtrusively. Internally, the library switches to offline mode, stopping synchronization between local and remote database. The local storage component enters the accumulating state.

Response measure

The internal switch to offline mode should be completely unnoticeable for the user. Besides a marking of resources which are not available in offline mode, the user should not have any discomforts caused by the loss of internet connectivity.

Stimulus

A user regains internet connectivity after using the web application offline.

Artifact

Complete JuSy library

Response

JuSy detects the regained internet connectivity. Next, the local storage component enters the resolving state, and the synchronization process is started. Finally, the synchronization process finishes, and the user is notified of eventual cancelled transaction in the reconciliation phase. The local and remote database are back in sync.

Response measure

Detection of internet connectivity should be no slower than 10 seconds.
The synchronization process should start immediately after the detection of the regained internet connection and data transfer should be as fast as the users internet connection allows.

Stimulus

A developer updates the remote database structure.

Artifact

Local storage component

Response

JuSy detects that the remote database has a different version number from the remote one and migrates the database structure accordingly.

Response measure

Regardless of the version of the local database there should be a migration path without data loss.

Testability

Stimulus

Developing the JuSy library.

Artifact

Complete JuSy library

Response

A developer must include unit testing for all his components.

Response measure

A minimum of 80% test coverage is required for each component.

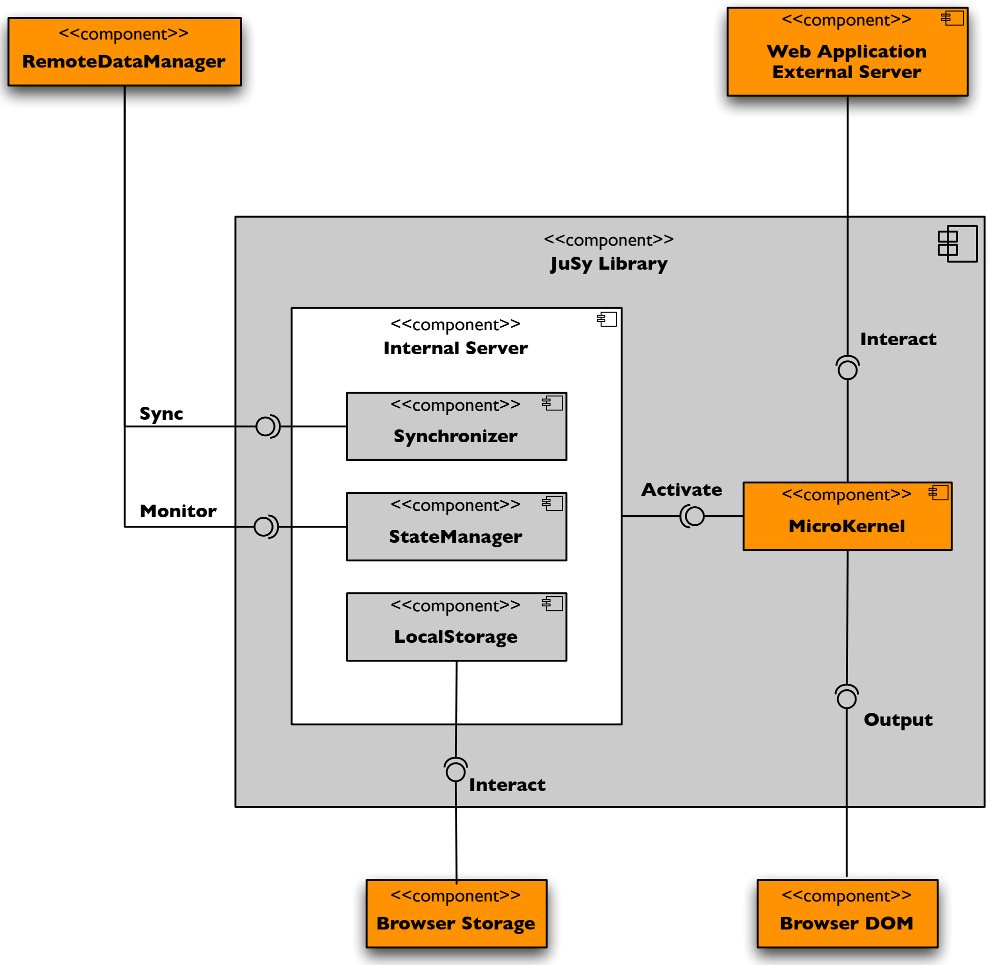
Chapter 6

Architecture

FIRST ITERATION

We start this chapter off by presenting a general view of the entire architecture, while introducing the main design pattern. Furthermore all main components will be identified and described using Class Responsibility Collaboration[17] (CRC) cards.

General component view



<<component>> RemoteDataManager
<<component>> Web Application External Server
<<component>> JuSy Library
<<component>> Internal Server
Interact
Sync
<<component>> Synchronizer
Monitor
<<component>> StateManager
Activate
<<component>> MicroKernel
<<component>> LocalStorage
Output
Interact
<<component>> Browser Storage
<<component>> Browser DOM

[17] For more information on this subject we refer the reader to the glossary

Main system pattern

Quality attribute : **Modifiability** - testability
Design pattern : **Microkernel**

As introduced in the project vision, *modifiability* is one of the most important quality attributes of the JuSy library. Because of this we have chosen a microkernel as the main system pattern : for systems which must be easily adaptable to changing requirements, it separates a minimal functional core from extended functionality and customer-specific parts.

When we look at a web application using JuSy as one system, we can view application-specific logic as an external server of the microkernel, while the core library functionality is implemented as components of the internal server. This also allows us to decouple the internal server components (i.e. local storage access, remote synchronization) so we can reuse, develop and test them separately.

The **microkernel** will :

- *Maintain system-wide resources* - Provide local data access to the external server and synchronizer.
- *Enable other components to communicate* - Allow the synchronizer to communicate with the local storage component, the local storage component with the state manager, etc.
- *Allows other components to access its functionality* - Provide output to the Document Object Model[18] (DOM), access to the library functionality for the external server, etc.

One might dispute that *performance* was another key quality attribute of our project. Microkernels typically incur a small performance penalty over a monolithic system, however we believe that the overhead introduced by the microkernel is irrelevant compared to other components. Moreover we plan to optimize the microkernel for fast local data access. Indeed, the performance goal was aimed at the local data access. If the synchronization process is slightly slower, this will not bother the user for two reasons : (1) the user will not notice it, because of the asynchronous nature, and (2) the lag introduced by synchronizing over an unreliable internet connection is of a far greater magnitude than that introduced by the microkernel.

[18] For more information on this subject we refer the reader to the glossary

Main components overview

Internal Server - Synchronizer	
Responsibilities	Collaborators
■ Add transaction info to UTQ	MicroKernel
■ Send UTQ	RemoteDataManager
■ Receive CancelSet	RemoteDataManager
■ Provide CancelSet	MicroKernel

Internal Server - StateManager	
Responsibilities	Collaborators
■ Determine browser connectivity status	MicroKernel “The Internet” Browser DOM
■ Determine remote server status	MicroKernel RemoteDataManager
■ Fire event in case of status changes	MicroKernel

Internal Server - LocalStorage	
Responsibilities	Collaborators
■ Allow unified database access through transactions	MicroKernel Browser Storage
■ Migrate database structure	MicroKernel Browser Storage
■ Determine the Read, Write and ResultSet of a transaction	MicroKernel Browser Storage

UNIVERSITEIT GENT

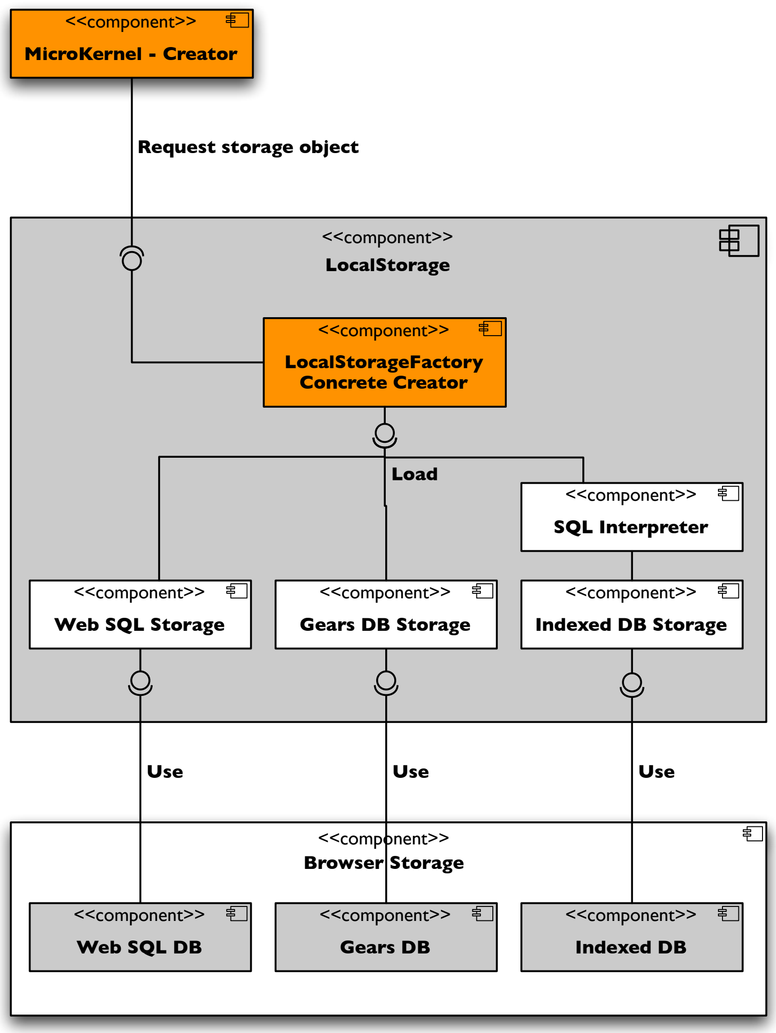
UPC

RemoteDataManager	
Responsibilities	Collaborators
■ Receive UTQ	Synchronizer
■ Process UTQ (preprocess and reconciliation)	Internal
■ Send CancelSet	Synchronizer
■ Provide database status information	StateManager

LocalStorage

The LocalStorage component is one of the elements of the microkernel's internal server. It provides unified access to storing data locally. Its main features are creating, updating and migrating a database, while allowing access to the database using a standard SQL dialect. These features have to be available regardless of the underlying browser, platform or storage technology.

Component diagram



Design pattern

Quality attribute : **Modifiability -** testability
Design pattern : **Factory**

We have chosen the factory pattern for the LocalStorage component, because it allows us to define an interface for creating an object, but let the subclasses decide which class to instantiate. We define an interface that specifies all the needed functionality, and allow underlying APIs to implement this functionality in their own way.

This allows us to use whatever storage technology is available in the browser, while also allowing adding new ways of storing locally easily. One simply has to implement the functionalities defined in the interface, and the new storage API will work with the library, without requiring any modifications to JuSy.

We have chosen to combine this pattern with reflection. The LocalStorageFactory will dynamically load all components that implement the LocalStorageInterface. All these components must be able to determine whether they are usable in the current browser and report this to the Factory, while also providing a priority. The priority determines a certain ordering amongst the different local storage components and assists the Factory in determining which one to use when more than one is available. We plan to allow an external server to overwrite this priority, to adapt to specific needs.

Components overview

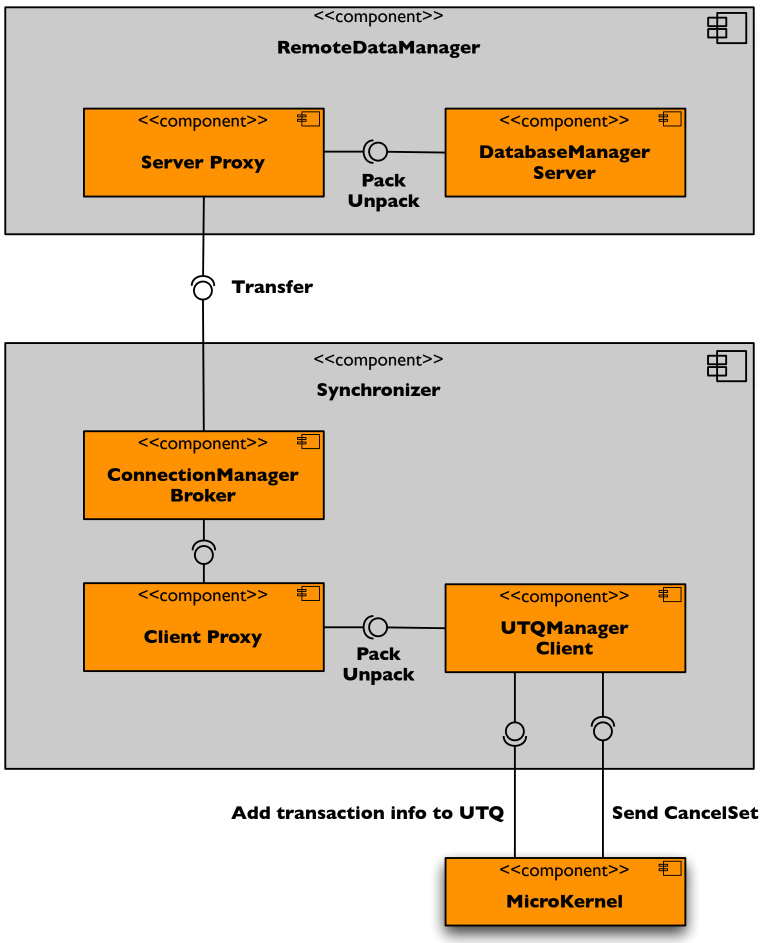
LocalStorageFactory	
Responsibilities	Collaborators
■ Create LocalStorageObject	MicroKernel Storage API implementations
■ Override priority	MicroKernel Storage API implementations

LocalStorage - Storage API implementation	
Responsibilities	Collaborators
■ Report availability on present browser	LocalStorageFactory
■ Report priority	LocalStorageFactory
■ Run transaction	MicroKernel
■ Process CancelSet	MicroKernel
■ Migrate database structure	MicroKernel

Synchronizer

This component provides synchronization between the remote database and the local browser database.

Component diagram



Design pattern

Quality attribute : **Modifiability**
Design pattern : **Broker**

We have chosen a broker system because we want decoupled components to interact through remote service invocations. Communication is coordinated by the broker component which forwards requests and relays results.

The proxies have been added to pack and unpack data, in our case, from and to JavaScript Object notation (JSON). Furthermore both the server-side and client-side proxy may need to split big data transmissions in smaller chunks to allow more efficient communication over an unreliable network, where it might be difficult to hold a sustained connection for a longer time.

Components overview

Broker Client - UTQManager	
Responsibilities	Collaborators
■ Add transaction Read, Write and ResultSet to UTQ	MicroKernel
■ Initiate synchronization process	ConnectionManager* MicroKernel
■ Provide CancelSet	ConnectionManager* MicroKernel

* Although UTQManager cannot communicate directly with the ConnectionManager, we have implied the usage of the client-side proxy for the sake of clarity and simplicity

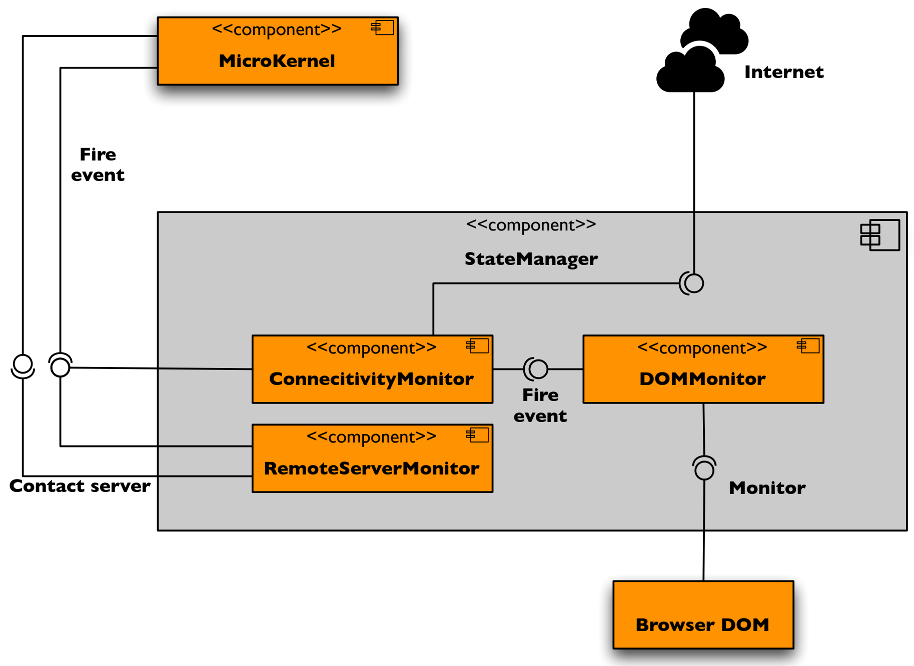
Broker Server - DatabaseManager	
Responsibilities	Collaborators
■ Process UTQ	ConnectionManager* Server database
■ Provide CancelSet	ConnectionManager*
■ Commit CommitSet	Server database

* Although DatabaseManager cannot communicate directly with the ConnectionManager, we have implied the usage of the server-side proxy for the sake of clarity and simplicity

StateManager

StateManager monitors three separate things : (1) the status of the remote server, (2) the browser status and (3) the internet connectivity status. Whenever one of these statuses changes, an event is sent to the MicroKernel, which in turn can then respond to the status change by invoking other internal server components.

Component diagram



Design pattern

Because of the relatively simple structure of this component we have chosen not to implement any specific design pattern.
Furthermore we have also chosen to let the DOMMonitor communicate directly with the browser DOM, although strictly this communication should be handled by the MicroKernel. We have chosen to do this because in JavaScript the DOM object is automatically available for every script, and it seems superfluous to introduce extra overhead by not using this JavaScript feature.

Components overview

DOMMonitor	
Responsibilities	Collaborators
■ Capture navigator.isOnline events	ConnectivityMonitor Browser DOM

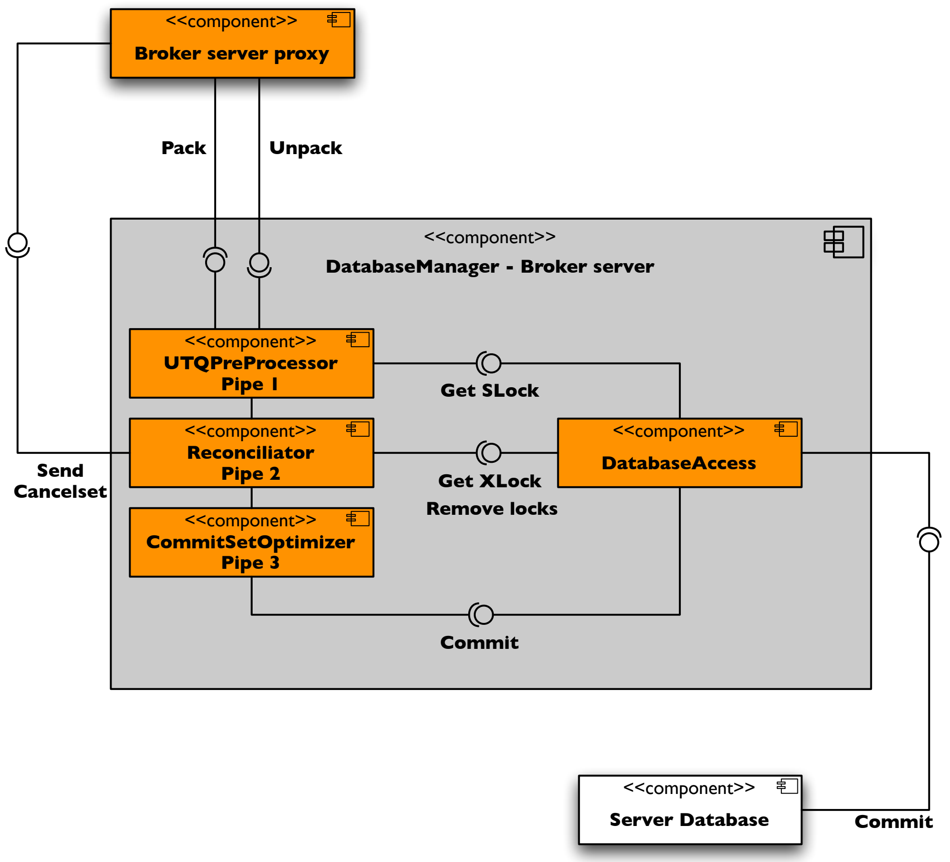
ConnectivityMonitor	
Responsibilities	Collaborators
■ Ping reliable internet server	“The Internet”
■ Respond to events	DOMMonitor
■ Fire event on connectivity status change	MicroKernel

RemoteServerMonitor	
Responsibilities	Collaborators
■ Verify remote server status	MicroKernel
■ Fire event on server status change	DOMMonitor

DatabaseManager

The DatabaseManager (broker server) is the component that takes care of most of the server-side logic. This includes accepting a UTQ from its client(s), processing this UTQ, updating the server database and sending the CancelSet to the client.

Component diagram



Design pattern

Quality attribute : **Performance** - modifiability - testability
Design pattern : **Pipes and filters**

We have chosen the pipes and filters pattern for two specific reasons.

- Firstly, since a UTQ received from a client can be considered a data stream, and the processing steps can be performed iteratively without requiring any knowledge of the following data, the performance can be improved tremendously by employing a pipeline system.
- Secondly, it allows us to develop and test the different steps of the pipeline independently, while also enabling us to add extra steps if this would prove necessary.

We have planned the following steps for the pipeline :

1. **Preprocessing -** Compute the ReadableSet of every transaction.
2. **Reconciliation -** Perform TLRSP[19] reconciliation.
3. **Optimizing -** Optimize the CommitSet generated by the TLRSP reconciliation algorithm to minimize the number of database writes.

Components overview

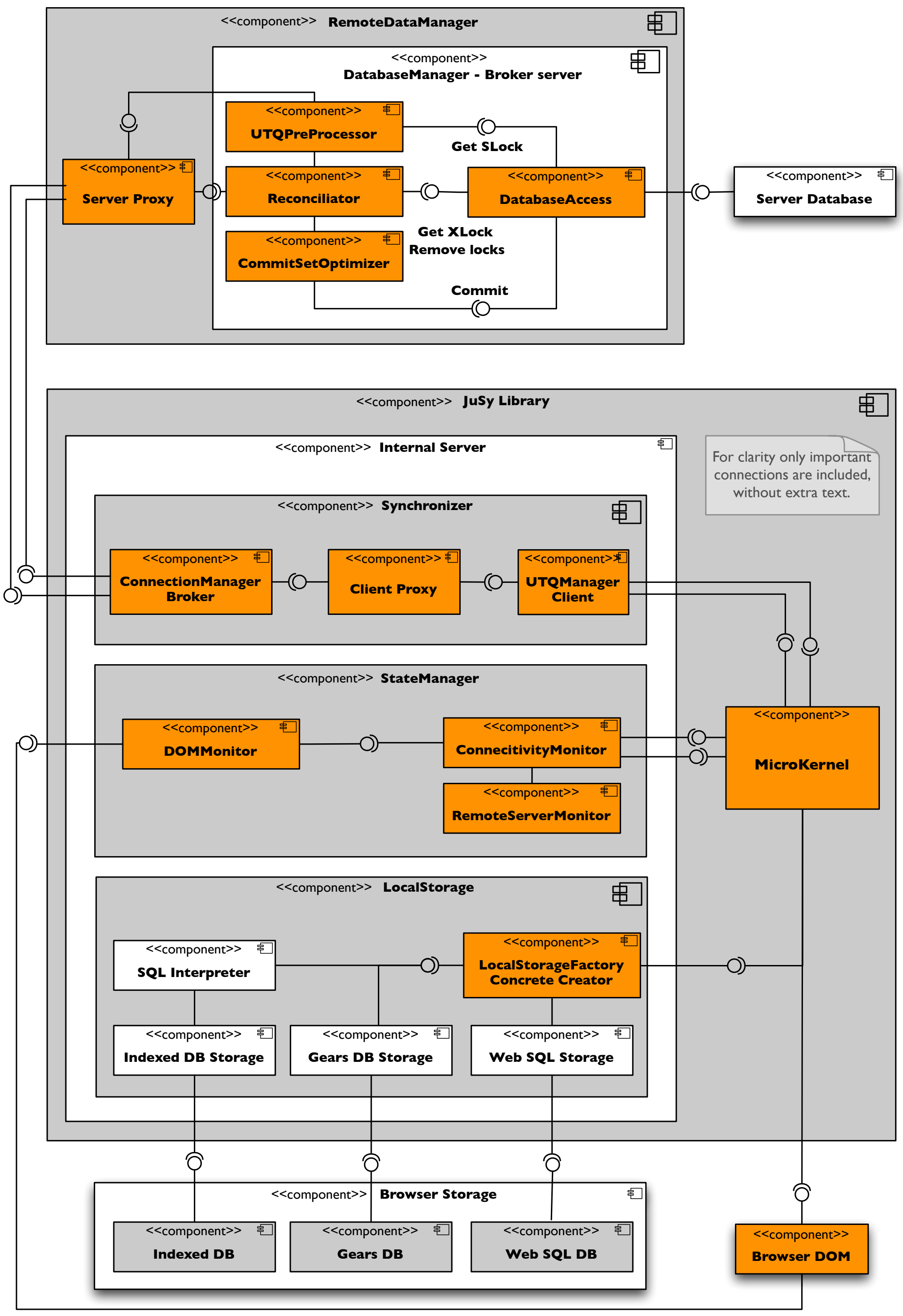
Pipe 1 - UTQPreProcessor	
Responsibilities	Collaborators
Accept next (or first) transaction in UTQ	Server proxy
Request a shared lock	DatabaseAccess
Compute ReadableSet of transaction	
Send ReadableSet	Reconciliator

Pipe 2 - Reconciliator	
Responsibilities	Collaborators
Accept next transaction + ReadableSet in UTQ	UTQPreProcessor
Request an exclusive lock	DatabaseAccess
Remove lock(s)	DatabaseAccess
Process transaction using TLRSP algorithm	
Send CommitSet	CommitSetOptimizer

[19] This algorithm is extensively described in the section "Database replication techniques" of chapter 2

Pipe 3 - CommitSetOptimizer	
Responsibilities	Collaborators
■ Accept next CommitSet	Reconciliator
■ Process CommitSet + wait for last CommitSet of UTQ	
■ Send optimized CommitSet	DatabaseAccess

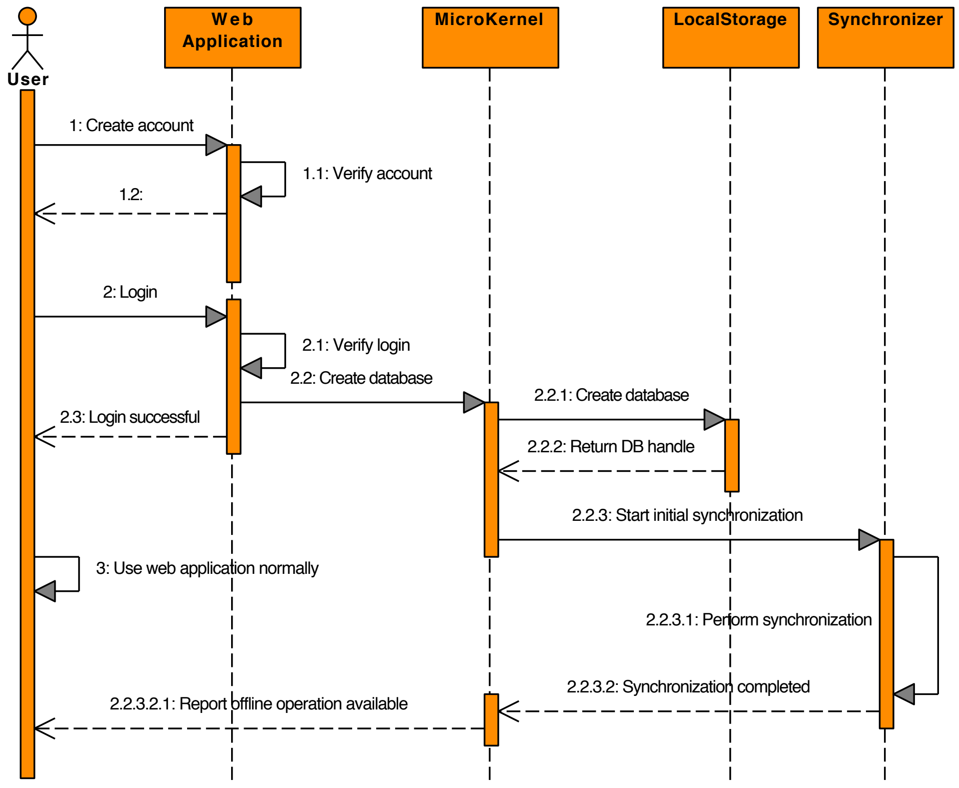
Full Architecture Component View


<<component>> RemoteDataManager
<<component>> DatabaseManager - Broker server
<<component>> UTQPreProcessor
Get SLock
<<component>> Server Proxy
<<component>> Reconciliator
<<component>> DatabaseAccess
<<component>> Server Database
<<component>> CommitSetOptimizer
Get XLock
Remove locks
Commit
<<component>> JuSy Library
<<component>> Internal Server
For clarity only important connections are included, without extra text.
<<component>> Synchronizer
<<component>> ConnectionManager Broker
<<component>> Client Proxy
<<component>> UTQManager Client
<<component>> StateManager
<<component>> DOMMonitor
<<component>> ConnecitivityMonitor
<<component>> RemoteServerMonitor
<<component>> MicroKernel
<<component>> LocalStorage
<<component>> SQL Interpreter
<<component>> LocalStorageFactory Concrete Creator
<<component>> Indexed DB Storage
<<component>> Gears DB Storage
<<component>> Web SQL Storage
<<component>> Browser Storage
<<component>> Indexed DB
<<component>> Gears DB
<<component>> Web SQL DB
<<component>> Browser DOM

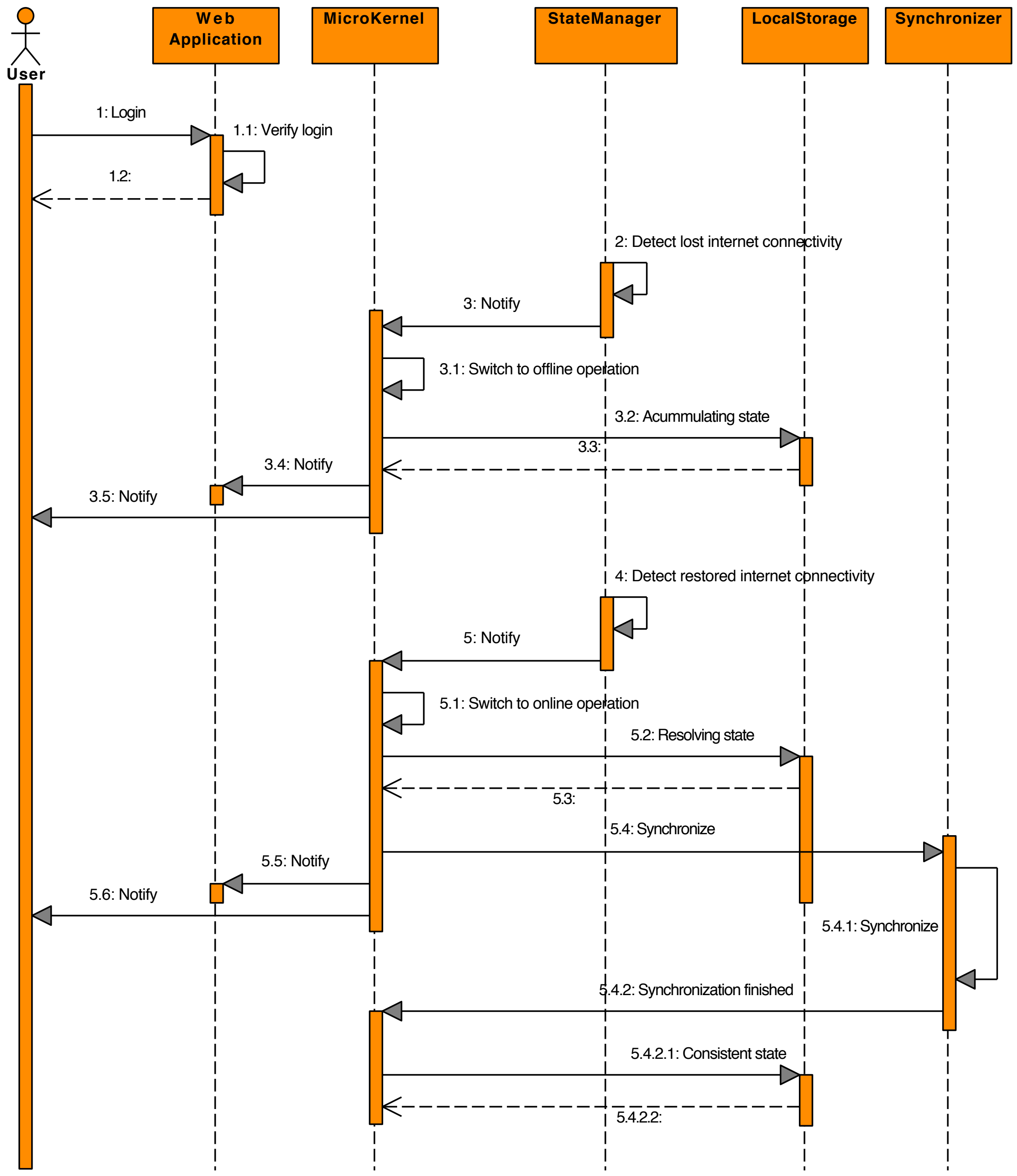
WHITEBOX SCENARIOS

For every applicable use case scenario defined in chapter 5, we will illustrate a whitebox scenario by means a of sequence diagrams in this section.

A user runs an application for the first time



Internet connectivity is lost while using a web application



Chapter 7
Planning & Economical analysis

INTRODUCTION

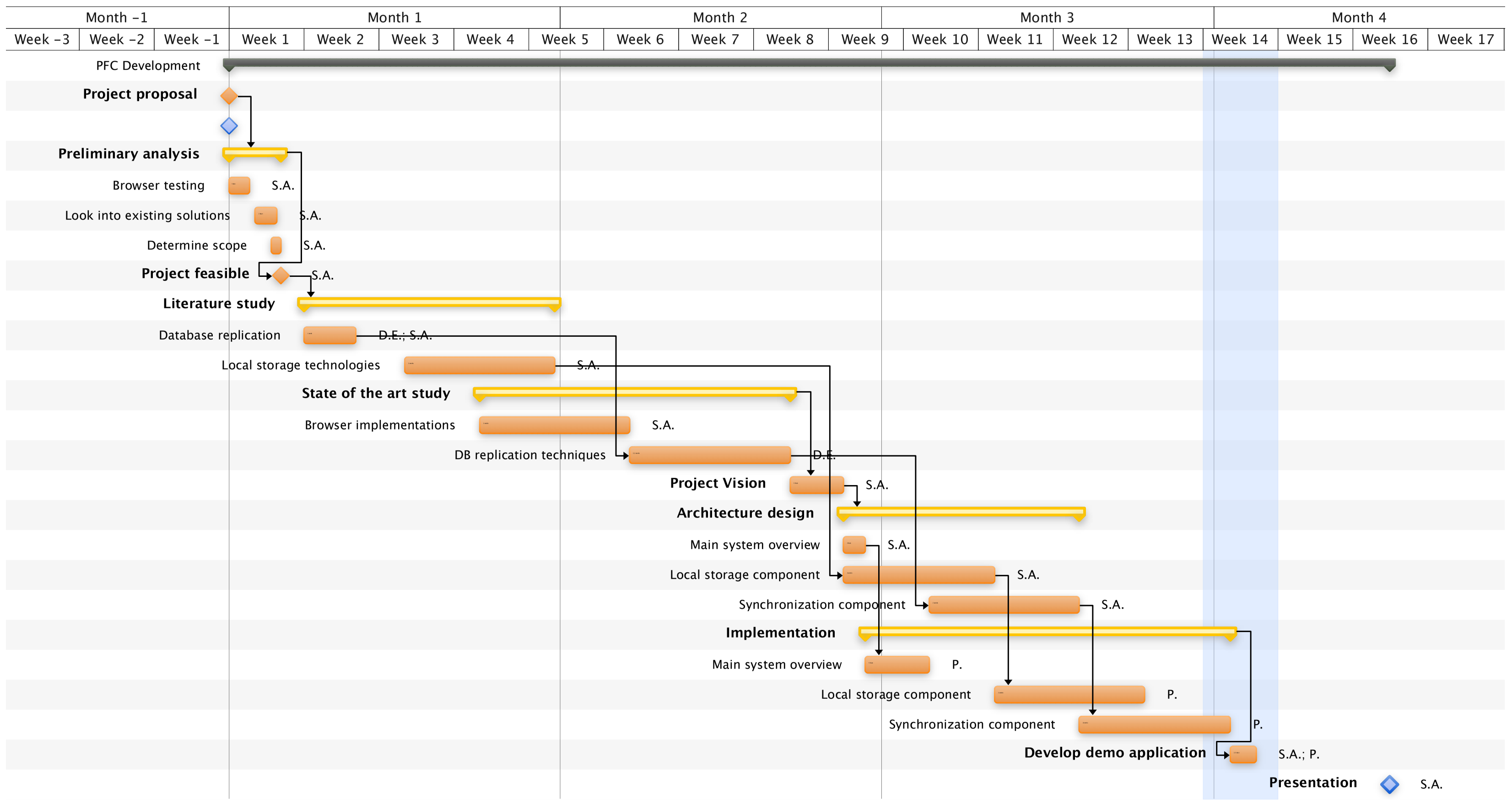
The goal of this chapter is to give the reader an idea of the amount of work that has gone into this project, and how much this would cost in a real-life situation.
In the first part we discuss the initial planning, demonstrating the hours spent to realize this project. We then determine the people needed to realize this planning, their estimated hourly rate, the needed external services and ultimately the total estimated cost of the project.

PLANNING

On the next page we have included a diagram illustrating our planning over the last 4 months per week. The project started of at the beginning of march 2010, and went on until the end of june 2010.

The reader will notice we have chosen to divide the planning into roughly 6 main phases :

- **Preliminary analysis -** Initial tasks we performed to determine the scope and feasibility of the project. This also included a glance at HTML5 specifications on web applications in general, and client-side browser database specifically.
- **Literature study -** Because of the required advanced database techniques it was necessary to dive into literature on databases. We started off with a broad review on database concepts, followed by writings describing database replication in general. This allowed us to classify the desired replication technique, after which we were able to study specific techniques fitting our project, in detail.
  Following this we started looking profoundly into ways of storing data locally. This consisted mainly of reading specifications and API descriptions.
- **State of the art -** Armed with the knowledge about database replication, we were able to critically look into existing middleware and decide why we needed to develop our solution.
  Secondly, we looked deeper into web browsers, and moreover, their implementations of state-of-the-art HTML5 features. Since it proved impossible to determine this based on information provided by browser builders, we developed a web application to test the features related to our project.
- **Project vision -** Although this might be a very short chapter, it defined the contours for the rest of the project. We wrote down our mission statement, decided on key features and determined the main quality attributes to evaluate the project.
- **Architecture design -** After setting up the project outline in the vision, we formalized this into a design-pattern driven architecture in this phase. We started off with use case scenarios and quality scenarios, after which we detailed the architecture with up to 3 iterations of architecture design.
- **Implementation -** Finally, in this phase we implemented the architecture, using agile, test-driven development.


Month -1
Month 1
Month 2
Month 3
Month 4
Week -3
Week -2
Week -1
Week 1
Week 2
Week 3
Week 4
Week 5
Week 6
Week 7
Week 8
Week 9
Week 10
Week 11
Week 12
Week 13
Week 14
Week 15
Week 16
Week 17
PFC Development
Project proposal
Preliminary analysis
Browser testing
S.A.
Look into existing solutions
S.A.
Determine scope
S.A.
Project feasible
S.A.
Literature study
Database replication
D.E.; S.A.
Local storage technologies
S.A.
State of the art study
Browser implementations
S.A.
DB replication techniques
D.E.
Project Vision
S.A.
Architecture design
Main system overview
S.A.
Local storage component
S.A.
Synchronization component
S.A.
Implementation
Main system overview
P.
Local storage component
P.
Synchronization component
P.
Develop demo application
S.A.; P.
Presentation
S.A.

ECONOMICAL ANALYSIS

In this section we will try to make an estimate of the total cost based on the aforementioned planning.
Firstly, we define which tasks are performed by which specialists.
In the second part we go over the required equipment and necessary external services.

Development costs

The following people would be involved in this project:

- **Software Architect -** This person will make the key decisions, determine the project vision, develop the architecture and generally manage the full project.
  Typical hourly rate[20] : 50,00 €

- **Database Expert -** This person will research the database replication technique, and assist the architect in making architectural decisions related to the database part.
  Typical hourly rate : 65,00 €

- **Programmer -** This person will implement the architecture.
  Typical hourly rate : 30,00 €

The next table shows the costs per specialist, and the total development cost :

Person	Hours	Rate	Total
Software Architect	408	50,00 €	20.400,00 €
Database Expert	84	65,00 €	5.460,00 €
Programmer	198	30,00 €	5.940,00 €
Total			31.800,00 €

Equipment and external services

Because of the nature of a web development project, testing is a very tedious job. Different browsers behave differently, and sometimes the same browser might behave differently on another platform. Therefore it is needed to run all tests on a plethora of browser/OS combinations. Especially when using agile development this can prove to be a very costly, time-consuming job.

[20] Based on average rates found on http://www.payscale.com

To solve this problem, services exist that automatically run your tests on all popular browser/OS combinations. One of the more popular services for this is crossbrowsertesting.com.

Service	Units	Rate	Total
Development computer	0,2	1.200,00 €	240,00 €
Xbrowsertesting.com pro account*	2	200,00 €	400,00 €
Total			640,00 €

* Monthly cost

Total cost

We can now calculate the project's estimated total cost :

Service	Total
Development	31.800,00 €
Equipment and external services	640,00 €
Total	**32.440,00 €**

Chapter 8

Conclusions & Future work

CONCLUSIONS

Having come at the end of a long journey, it is time to look back at the beginning. We stated that we want to bring the performance of native applications to web applications, and on top of that allow them to operate without a working internet connection.

We believe that this project might appeal to both end users and developers. End users, because of the goals stated above, and developers because they do not have to go through the tedious job of implementing database replication and a database abstraction layer.

Our architecture was designed while keeping our main quality attributes in mind :

- **Modifiability -** Because of the use of a microkernel as main system pattern, the internal server components are very loosely coupled and easily reusable on their own. A developer that only needs synchronization for his project, could easily combine our synchronizer-component with his own local database implementation.
  Additionally, because of the use of a factory, in combination with reflection, for the local storage component, new and future local storage technologies can easily be added to the library without any modifications to the library itself.
- **Usability -** Developers making use of this library are completely shielded from the details of database replication, local storage, online/offline detection, etc. From the outside JuSy actually behaves like a normal database that can be queried in a normal way.
  Furthermore, the added value for end users to be able to use their web application without an active internet connection(especially for mobile users) far surpasses the eventual extra trouble of having to let the application finish its synchronization process, or update their browser.
- **Performance -** Although the primary goal was to improve a web application's performance, we made some small tradeoffs while developing the architecture (since the use of a microkernel introduces a small overhead). The local storage and synchronization components however, have been designed to perform as fast as possible.
- **Testability -** By decoupling all major components from each other they can easily be tested separately.

In conclusion, we believe we have created a versatile architecture to suit both the needs of web developers and end users.

Future work

Security

On more than one occasion we realized that some applications simply cannot be allowed to replicate data to a user's computer. Online banking applications for example, come to mind. The reason for this is that a user cannot be trusted in keeping his system 100% secure at all times.
The first trivial reason for this is that the average computer user does not always keep his system up-to-date, and is thus vulnerable to data theft. A second, less trivial reason is the physical security of a computer. People lose computers on occasion, they get stolen, etc.
An interesting angle to this might be to implement an encryption layer on top of the database layer. This would assure that the locally saved data is unusable when it fell in the wrong hands.

During the development of this project, we knowingly set aside the security aspects. We chose to do so, because solving the security issues concerning replicating server data to a user is a problem far beyond the scope of this project. We believe this might even be the subject of a full, new master's project.

Performance

When we looked into ways of saving data locally, we only looked at the feature set and the availability in popular browsers. An interesting addition to this might be to compare the different local storage technique's performance. This way a more intelligent choice could be made when more than one way of saving data locally is available.

Another interesting path to go down would be to research alternatives for JavaScript. Although this is theoretically the only way to run program logic at the client side (without resorting to plug-ins, e.g. Flash, SilverLight, Java Webstart, etc. or proprietary technologies, e.g. ActiveX), some APIs have been built on top of JavaScript. One promising example of this is Objective-J, an Objective-C emulation for browsers. It features an identical syntax, is fully object-oriented (contrary to JavaScript itself, where several workarounds are necessary to support the OO paradigm) and even features a GUI toolkit.

Because of the vast amounts of research put in these kinds of projects, they have been highly optimized. Because of this we believe they might perform better than plain JavaScript, even though they require an extra layer of abstraction.

Coupling with MVC frameworks for web applications

During the last few years, Model-View-Controller[21] (MVC) frameworks have gained enormous popularity amongst web developers. Popular examples include Ruby on Rails (RoR), Zend framework, Spring framework, etc.

The advantages of using such a framework are numerous :

- Strict separation of business logic and presentation, allowing developers and graphical designers to work separately.
- Tried-and-tested foundation for a web application.
- Common web applications components are usually already implemented, well-tested and very feature-complete.
- Easier to reuse components.
- Object relational mapping[18] (ORM) library usually built in.

Although all these features are equally valid and remarkable, the last one is most interesting for our project. An ORM library typically creates database tables that are bound specifically to the used framework. They often employ complicated algorithms, which mostly result in database table structures that make no sense outside of the used ORM library. This also makes it very difficult to use this project on data under ORM management.

An extra addition to this project would be to adapt the algorithms used by an ORM library and port them to the client side. This way, a developer would be able to interact in the same way with the data in his client-side code, as he would in his server-side code. Even more interesting might be to develop an “ORM specification”, which could be adopted by framework builders and client-side library builders.

[21] For more information on this subject we refer the reader to the glossary

Glossary of terms

ACID	Atomicity, Consistency, Isolation and Durability. ACID is a set of properties that guarantee database transactions are processed reliably and transform the database from one consistent state to another.
AJAX	Asynchronous JavaScript and XMLHttpRequest. Ajax is a group of interrelated web development techniques used on the client-side to create interactive web applications. With Ajax, web applications can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.
API	An Application Programming Interface (API) is an interface implemented by software which enables it to interact with other software. It is similar to the way the user interface facilitates interaction between humans and computers.
Browser DOM	The Document Object Model (DOM) is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. In the contest of browsers, DOM is used internally by the layout engine to represent a webpage.
CRC Cards	Class Responsibility Collaboration (CRC) cards are a brainstorming tool used in the design of object-oriented software. They are typically used when first determining which classes are needed and how they will interact and contain : the class name, its Super and Sub classes (if applicable), the responsibilities of the class and the names of other classes with which the class will collaborate to fulfill its responsibilities.
DBMS	A Database Management System (DBMS) is a set of computer programs that controls the creation, maintenance, and the use of a database. Popular examples are Microsoft Acces, Oracle, PHPMyAdmin, etc.
HTML	HyperText Markup Language (HTML) is the predominant markup language for web pages. It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes, and other items.

JSON	JavaScript Object Notation (JSON) is a lightweight text-based open standard designed for data interchange. It is derived from JavaScript for representing simple data structures, arrays and objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for virtually every programming language.
MVC	Model–View–Controller (MVC) is an architectural pattern used in software engineering. The pattern isolates the application logic from user input and presentation, permitting independent development, testing and maintenance of each.
ORM	Object Relational Mapping (ORM) libraries hide the raw database accesses, by allowing the developer to create native objects, which are automatically persisted by the library. The developer only needs to define his objects, and the relations amongst them, while the ORM library takes care of the table creation, structure, queries, etc.
SQL	SQL (Structured Query Language) is a database computer language designed for managing data in relational database management systems, and is originally based upon relational algebra. Its scope includes data query and update, schema creation and modification, and data access control.
W3C	The World Wide Web Consortium is an international community where member organizations, a full-time staff, and the public work together to develop Web standards. Led by Web inventor Tim Berners-Lee and CEO Jeffrey Jaffe, W3C's mission is to lead the Web to its full potential.
WHATWG	The Web Hypertext Application Technology Working Group was founded by individuals of Apple, the Mozilla Foundation, and Opera Software in 2004. Today it is a growing community mainly focused on the development of HTML 5.
XHTML	XHTML (Extensible Hypertext Markup Language) is a family of XML markup languages that mirror or extend versions of the widely used Hypertext Markup Language(HTML), the language in which web pages are written.

References

[1] World Wide Web Consortium (December 24, 1999), 'HTML 4.01 Specification',
http://www.w3.org/TR/html401/

[2] ECMA International (December 2009), 'ECMAScript Language Specification',
http://www.ecma-international.org/publications/standards/Ecma-262.htm

[3] World Wide Web Consortium 'Definition of origin',
http://www.w3.org/TR/html5/browsers.html#origin-0

[4] ——— (January 5, 2010), 'Indexed Database API',
http://www.w3.org/TR/IndexedDB/

[5] Gears Team Ian Fetter 'Hello HTML5', February 19, 2010
http://gearsblog.blogspot.com/2010/02/hello-html5.html

[6] Justin James (May 6, 2010), 'interview HTML 5 Editor Ian Hickson discusses the spec's current status',
http://blogs.techrepublic.com.com/programming-and-development/?p=2524&tag=rbxccnbtr1

[7] SQLite Consortium 'SQL As Understood By SQLite',
http://www.sqlite.org/lang_transaction.html

[8] World Wide Web Consortium 'HTML5 - Browser state',
http://dev.w3.org/html5/spec/offline.html#browser-state

[9] Nickolay Ponomarev (January 9, 2010), 'Mozilla Developer Center - Online and offline events',
https://developer.mozilla.org/en/Online_and_offline_events

[10] Bernstein, Philip A., Vassos Hadzilacos, and Nathan Goodman (1987), *Concurrency Control and Recovery in Database Systems*, (Addison Wesley Publishing Company).

[11] Breitbart, Yuri, Dimitrios Georgakopoulos Marek Rusinkiewicz, and Abraham Silberschatz 'Providing high availability using lazy replication', *IEEE Transactions on Software Engineering (TSE)*, 17 - 9 954-60.

[12] Ladin, R., et al. (1992), 'Providing high availability using lazy replication', *ACM Transactions on Computer Systems*, 10 - 4 360-91.

[13] Ladin, R., B. Liskov, and L. Shrira (1990), 'Providing high availability using lazy replication', *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, 10 - 4
43-57.

[14] Yasushi, Saito and Shapiro Marc (2005), 'Optimistic replication', *ACM Computing Surveys*, 37 - 1 42-81.

[15] Gray, Jim, et al. (May 1996), 'The dangers of replication and a solution', *Proceedings of the 1996 ACM SIGMOD Conference*, 173-82.

[16] Hara, Takahiro and Sanjay Kumar Madria (July 2009), 'Consistency Management Strategies for Data Replication in Mobile Ad Hoc Networks', *IEEE Transactions on Mobile Computing*

[17] Breitbart, Yuri and Henry F. Korth (1997), 'Replication and consistency: being lazy helps sometimes', *Symposium on Principles of Database Systems*, 173-84.

[18] Zhiming, Ding, Meng Xiaofeng, and Wang Shan (July 2002), 'A Transactional Asynchronous Replication Scheme for Mobile Database Systems', *J. Computer Science & Technology*, 17 - 4

[19] Brady Eidson (October 19, 2007), 'WebKit Does HTML5 Client-side Database Storage', http://webkit.org/blog/126/webkit-does-html5-client-side-database-storage/

[20] Jeremy Orlow 'IndexedDB Design Documentation', http://sites.google.com/a/chromium.org/dev/developers/design-documents/indexeddb

[21] ——— 'Baby steps towards IndexedDB: Start implementing callbacks', March 9, 2010 https://bugs.webkit.org/show_bug.cgi?id=35911

[22] Mozilla Foundation (January 10, 2007), 'Gecko 1.9 roadmap', https://wiki.mozilla.org/Gecko_1.9_Roadmap#Web_app_deployment_and_capability_improvements

[23] Shawn Wilsher (May, 24 2010), 'Mozilla Developers Wiki - Indexed Database', https://wiki.mozilla.org/Firefox/Projects/IndexedDB

[24] Mike Beltzner 'Early product plan for Firefox 4', May 10, 2010 http://beltzner.ca/mike/2010/05/10/firefox-4-fast-powerful-and-empowering/

[25] III Scott M. Fulton (December 22, 2009), 'The once and future king: Test build of Opera crushes Chrome on Windows 7', http://www.betanews.com/article/The-once-and-future-king-Test-build-of-Opera-crushes-Chrome-on-Windows-7/1261519843

[26] Opera Software (April 15, 2010), 'Opera version history', http://www.opera.com/docs/history/#facts

[27] ——— (March 10, 2010), 'Web specifications supported in Opera Presto 2.4', http://www.opera.com/docs/specs/presto24/#database

[28] Adrian Bateman (March 9, 2010), 'Working with the HTML5 Community', *Indexed DB Proposal*,

http://blogs.msdn.com/b/ie/archive/2010/03/09/working-with-the-html5-community.aspx

[29] Pablo Castro (January 4, 2010), 'HTML5 does databases', http://blogs.msdn.com/b/pablo/archive/2010/01/04/html5-does-databases.aspx

[30] ——— (January 5, 2010), 'Want to work on HTML5 databases? We're looking!', http://blogs.msdn.com/b/pablo/archive/2010/01/05/want-to-work-on-html5-database-stuff-we-re-looking.aspx

UNIVERSITEIT GENT

UPC