TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MIKA VÄLIMÄKI

DATA SYNCHRONIZATION IN WEB-BASED LIQUID SOFTWARE SOLUTIONS

Master of Science thesis

Examiner: Prof. Kari Systä
Examiner and topic approved by the
Faculty Council of the Faculty of
Information Technology
on 6th April 2016

ABSTRACT

MIKA VÄLIMÄKI: Data synchronization in web-based liquid software solutions
Tampere University of Technology
Master of Science thesis, 59 pages, 0 Appendix pages
May 2016
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Kari Systä
Keywords: Liquid software, Web-based, Synchronization, Cross-platform, JavaScript, Push API, Service worker, Web browser, IndexedDB

When a computer application is designed with a liquid software approach it means that the application can be used on multiple devices and on multiple environments. In addition the user can move from one device to another and the state of the application seamlessly transfers when switching between devices. It also means that the persistent data used by the program is synchronized between the devices. Using the program on multiple devices can either be sequential or simultaneous and there can be more than one user using the same application.

In this thesis the liquid software approach was tested in a web application to see what kind of challenges the browser environment would have. The focus of the application was on synchronizing persistent data between devices. The web application approach was chosen because it can be run on all platforms that have support for a modern browser. In order to test it a proof of concept application was made. The requirements for the application were that it would not require third party add-ons and that the core design would not be tied to any third party system. Different technologies were tested and according to the tests the three most important technologies were Push API, service workers and IndexedDB. They all are new web technologies and their standardization is not yet finished. The standards are being developed by W3C, WHATWG and IETF.

Persistent data synchronization that satisfies the requirements of a liquid software is possible to develop for the browser environment. Web browsers still face challenges such as that the browser does not have access to low level services of the underlying operating system which for some applications would be mandatory. Also if the liquid software approach is to be combined with an Internet of things application then the web-runtime might not be possible to be used with all of the devices.

TIIVISTELMÄ

MIKA VÄLIMÄKI: Datan synkronointi verkkopohjaisissa liquid software-sovelluksissa
Tampereen teknillinen yliopisto
Diplomityö, 59 sivua, 0 liitesivua
Toukokuu 2016
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastajat: Prof. Kari Systä
Avainsanat: Liukkaat järjestelmät, Verkkopohjainen, Synkronointi, Monialusta, JavaScript, Push API, Service Worker, Verkkoselain, IndexedDB

Liukkaat ohjelmistot ovat tietokoneohjelmia, joita voi käyttää useassa eri ympäristössä usealla eri alustalla joko vuorotellen tai samanaikaisesti. Näiden ominaisuuksien lisäksi laitteelta toiselle siirryttäessä aiemman ohjelman tila ja tiedot on jo valmiiksi synkronoitu, kun käyttäjä ottaa uuden laitteen käyttoonsä. Ohjelman käyttö eri laitteilla voi olla joko vuorottaista tai samanaikaista, ja käyttäjiä voi olla yksi tai useampi.

Tässä opinnäytetyössä liukkaiden ohjelmistojen suunnittelutapaa testattiin verkkoselainten ympäristössä, jotta nähtäisiin millaisia ongelmia se nostaa esiin. Työssä keskityttiin pysyvän datan synkronointiin eri laitteiden välillä. Verkkoympäristö valittiin siitä syystä, että niitä voi ajaa jokaisella alustalla, jolla voi ajaa modernia verkkoselainta. Testiä varten luotiin näytesovellus, jonka vaatimuksena oli riippumattomuus kolmannen osapuolen tuotteista. Erilaisia teknologioita kokeiltiin ja lopulta liukkaan ohjelmiston toteutukseen löytyi kolme tärkeää selainteknologiaa: Push API, service worker ja IndexedDB. Nämä kaikki ovat uusia työkaluja, eikä niiden minkään standardointi ole vielä valmis. Standardeja kehittävät järjestöt W3C, WHATWG ja IETF.

Liukas ohjelmisto on mahdollista toteuttaa selaimessa, mutta selaimen omat rajoitukset voivat joissain tapauksissa olla liikaa. Selain ei pääse käsiksi käyttöjärjestelmänsä matalan tason palveluihin, mikä voi olla joillekin sovelluksille todella tärkeää. Jos liukkaat ohjelmistot halutaan yhdistää esineiden Internettiin, täytyy ottaa huomioon, ettei selainympäristö ole ajettavissa kaikilla laitteilla.

PREFACE

The subject of this thesis was given by Prof. Kari Systä and Prof. Tommi Mikkonen from the Department of Pervasive Computing at Tampere University of Technology. I would like to thank both of them for guiding and examining my thesis. I would also like to thank Juho Leppämäki and Mika Torhola from Atostek Oy for guiding me with my thesis.

Lastly, I would like to thank my parents, my brother and my fiancée for all their encouragement and support.

Tampere, May 18th, 2016

Mika Välimäki

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

$HTML$	HyperText Markup Language
CSS	Cascading Style Sheets
$W3C$	The World Wide Web Consortium
$WHATWG$	The Web Hypertext Application Technology Working Group
$IETF$	Internet Engineering Task Force
IE	Internet Explorer
WP	Windows Phone
OS	Operating system
PC	Personal computer
API	Application Programming Interface
SQL	Structured Query Language
I/O	Input/Output
DOM	Document Object Model
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
SQL	Structured Query Language
$RDBMS$	Relational Database Management System
OOP	Object-Oriented Programming
ORM	Object-Relational Mapping
$OODBMS$	Object Oriented Database Management System
XHR	XMLHttpRequest
$HTTP$	Hypertext Transfer Protocol
$HTTPS$	HTTP Secure
$JSON$	JavaScript Object Notation

1. INTRODUCTION

1.1 Background

Nowadays it is normal to have an Internet connection almost anywhere in the western world. It is also common to own a smart phone and a tablet PC in addition to a desktop PC and a laptop. However, smart devices are getting more common in other areas of life as well: fridges, washing machines, watches, cars and so on.

One can easily develop software for one of these devices but problems arise when the software needs to be run on another device and each device has its own type of OS. There might be a need to share the state of the software between the different devices which can be difficult in itself but with different OS's it could be even more challenging. Even if two devices had the same OS then there could still be some compatibility issues if the device manufacturers are different or if the OS editions differ from each other. This problem can be avoided by using a web browser as the platform of the software since modern browsers implement many of the new HTML5 standards very well which gives the browsers many OS like features.

When the user owns multiple devices that all have a certain software installed it would be useful if they could use the software with any of the devices at any time depending on the situation. It would also be useful if the state and data was transferred effortlessly and quickly from device to device. This is liquid software - an approach that will allow data and applications to seamlessly move between multiple devices and screens[51].

1.2 Aim of the thesis

In this thesis I researched how to implement persistent data synchronization for a liquid software application that is developed with JavaScript and that can be run on a modern web browser. The aim is to keep the persistent data of the application

synchronized between several devices. Two important aspects of the data synchronization are speed and offline support. The data must be synchronized quickly when modifications are done. If the user has been offline while modifying data then the data should be synchronized when the user connects back to the Internet.

For this thesis I have developed a proof of concept application along with data synchronization modules. The goal of the data synchronization modules is to offer an easy API which will take care of synchronizing persistent data between the browser and the server. Issues regarding simultaneous modification of data by multiple devices has been omitted from the scope of this thesis. The application is written with JavaScript which will make use of the standard features of HTML5 such as IndexedDB, WebSocket and Push API. I will use Node.js [19] to implement the server side of the application. I will discuss these technologies later.

1.3 Structure of the thesis

This document is structured as follows: Chapter 2 discusses the theory of liquid software and data synchronization and their challenges. Chapter 3 discusses the overall technologies related to data synchronization and the different web technologies that the HTML5 standard has to offer for modern browsers. Chapter 4 focuses on the principles of the data synchronization solutions while Chapter 5 focuses more on the implementation of the proof of concept application. The evaluation of the chosen solutions is presented in Chapter 6. Outlooks of the future are discussed in Chapter 7 and the conclusion is in Chapter 8.

2. LIQUID SOFTWARE

2.1 Description

In a liquid software approach the applications and data can flow from one device or screen to another seamlessly allowing the users to roam freely between their devices. It is not a single technology but instead it is a mindset supported by a specific set of technologies.[51]

Liquid software divides the concept of data in persistent data and to the state of the application. Persistent data would be something that is usually stored in databases. The state of the application contains the information about the current view of the application, the possible configurations made by the user and other information related to state.

Liquid software has three categories: sequential screening, simultaneous screening and collaboration scenario. In sequential screening a single user runs an application on different devices at different times and each time the user switches to a new device the current state and the data of the application is transferred to the new device. In simultaneous screening the user uses multiple devices at the same time and the view of the application can be changed based on the device. In a collaboration scenario several users run the same application on their devices and the collaboration can be either sequential or simultaneous.[31]

2.1.1 Requirements

According to the Liquid Software Manifesto [51] the key requirements for liquid software are the following:

1. Effortless roaming: “In a truly liquid multi-device computing environment, the

users shall be able to effortlessly roam between all the computing devices that they have."

2. Usability: "Roaming between multiple devices shall be as casual, fluid and hassle-free as possible; all the aspects related to device maintenance and device management shall be minimized or hidden from the users."

3. Data synchronization: "The user's applications and data shall be synchronized transparently between all the computing devices that the user has, insofar as the application and data make sense for each device."

4. State synchronization: "Whenever applicable, roaming between multiple devices shall include the transportation / synchronization of the full state of each application, so that the users can seamlessly continue their previous activities on any devices."

5. Freedom of ecosystems: "Roaming between multiple devices shall not be limited to devices from a single vendor ecosystem only; ideally, any device from any vendor should be able to run liquid software, assuming the device has a large enough screen, suitable input mechanisms, and adequate computing power, connectivity mechanisms and storage capacity"

6. Control: "The user shall remain in full control regarding the liquidity of applications and data. If the user wishes certain functionality or data to be accessible only on a single device, the user shall be able to define this in a simple and intuitive fashion."

In addition to the previous requirements regarding this thesis the application must be able to run in a browser. Full cross-browser compatibility is not required in terms of this thesis so the requirements are met if the application can run in at least one modern browser as long as it is not dependent on any browser add-ons.

2.1.2 Current obstacles and solutions

Since liquid software must be able to run on multiple devices the main obstacle is the development of the application for different environments. Android [13], iOS [6], OS X [7], Linux [25] and Windows [59] have their own ecosystems with their multiple popular versions. Version control and maintenance gets increasingly hard

with multiple environments because the developers have to implement the same functionalities in a specific way for each environment.

The need to simplify development for multiple devices has been noted and products such as PhoneGap [1], Qt [45] and Apache Cordova [5] have emerged on the market. These products aim to give the developers a single programming language to use and simple APIs to implement the software product across different environments.

Another approach is to move the software into a web-runtime which means that the application can run in a browser. All modern devices and operating systems have at least one modern browser which implements most of the HTML5 features and different standards defined by the W3C [68]. JavaScript and HTML along with the browsers have improved vastly over the years and they are now very potent platforms for a client software. In some cases different browsers force the developers to implement certain features in multiple different ways but the overall environment is mostly constant.

The biggest obstacles for browsers are system calls, I/O handling and offline usage. With JavaScript it is not possible to handle local files so the assistance of the server is required. It is usually not possible to communicate with the I/O ports of a computer with the browser without some 3rd party add-on. LocalStorage [66] and cookies of a web browser can store only key-value pairs where the value is of type string and their storage size is very limited. W3C is currently working on new standards for better ways to store data and some new versions of certain browsers already implement these features. These standards are AppCache, IndexedDB and ServiceWorker which will be discussed later.

2.2 Current examples

Examples of well-known applications that can be seen somewhat liquid are Telegram [52], Google Mail [12], Amazon's Whispersync [2] and Xbox Smartglass [28]. All of them can be run on multiple devices and either their persistent data or the state of the application can be synchronized between the devices.

Telegram is a chat software that can be run on Android, iOS, Windows Phone [30], browser, Windows, OS X and Linux [52]. If the user has multiple instances open and they all belong to the same user then any message sent or received in any of

the devices will update the message data on all devices given that all of them are connected to the Internet. On Telegram the data is synchronized but the state of the application is not. If the user navigates to a certain chat room on one device then the other devices remain in the view they are in. If the user opens a chat room which has a new message on one device then the notification icon of a new message will be removed from all other devices.

Google Mail is an email service which can be used with a web browser, an Android device or an iPhone [12]. Whenever user receives email to their account it is seen on all active devices. There are some delays between the updates. If user deletes a message it can still be seen on other instances for a while until it disappears.

Amazon's Whispersync is software that can be found in the Fire and Kindle devices of Amazon and from Kindle reading applications. Whispersync allows the user to view the bookmarks, highlights, and notes that were created on other devices registered to the same Amazon account. Additionally Whispersync automatically synchronizes the bookmarks and the annotations of the user. [2]

With Xbox SmartGlass, the user can connect their mobile device to their Xbox 360 or Xbox One console. The user can then control the console with their mobile device or use the mobile device as a second screen while playing a video game or while watching a movie. Xbox SmartGlass has different liquid software type features when compared with the other examples. In other examples the persistent data was synchronized between the devices but in SmartGlass the focus is on synchronizing the state of the application in real-time. The mobile device can be used when watching TV, listening to music or playing video games on the Xbox console and the view of the second screen changes based on the use case. [28]

All these examples can be considered to be almost liquid software since they implement some key features of the liquid software manifesto presented in 2.1.1 but they are lacking either in the fluid roaming between the devices by having too long delays in synchronization or they only synchronize the persistent data but not the state of the application or vice versa. The examples do not give the user much or any control over the liquidity of the software and instead the amount of data that is synchronized is either fixed or it can only be toggled on or off. The fact that some of these examples are tightly locked in a single vendor ecosystem or have support for only a limited amount of devices is also preventing them from being considered fully liquid.

2.3 Benefits of liquid software philosophy

In this section I will discuss what kind of impact the liquid software design philosophy could have on some general software applications and their use cases in a work environment. The use cases are divided based on the point of view of each liquid software category. Each point of view focuses on a different worker with different tasks.

2.3.1 Sequential screening

In sequential screening the user has different devices and they use one device at a time depending on time and place. This aspect of liquid software design philosophy would fit really well in a scenario of a worker with four different work locations: their own desktop at the office, a faraway work site, the journey between the office and the work site, and finally meeting rooms at the office. The worker manages the work sites and assigns different tasks for other workers and reports the progress of the work site for the other managers at the office.

In a liquid use case the worker would have an enterprise resource planning tool which has support for multiple platforms and the data it generates would be integrated with their note making application and reporting application. At the office they could have a desktop PC on which they could list the work sites and the tasks of the sites or gather more information from them through reports. If the worker is free to use a mobile device during the journey to the work site, the worker could further gather information about the work site or assign tasks beforehand. At the work site the tasks could be verified or additional information could be gathered through a mobile device and real-time reports could be written already at the site. The work site could even have many different devices and each of them would be synchronized with the necessary working data. The day's work and tasks status would be easy to update at the site and reviewed later at the office and presented in possible meetings.

The same task without liquid philosophy could be closer to a situation where the user has a single desktop PC with an enterprise resource planning tool which has support for only one device at their office workplace and the rest of the information is handled with phone calls or notebooks. When the user gets back from the work site and wants to update reports or the work progress related documents they would

have to manually type in the information that resides in their own memory or in the notebooks.

Full liquidity regarding this case is not completely necessary as the tasks can be done with different devices to different server backends almost as easily as in the liquid approach. Also the data synchronization speed does not have to reach the liquid manifesto level of fluidity as long as the time frame between switching devices is long enough.

2.3.2 Simultaneous screening

In simultaneous screening the user has different devices which are used at the same time and they all have their necessary data constantly synchronized. Imagine a line of work where the worker modifies a physical object with different tools. One example of this kind of work could be an assembling of a car or other vehicle but the example that is presented here could work with any similar project. In a typical scenario the worker would work on the object with their devices and they would estimate the completeness of the work based on their previous experiences and their expertise. In a somewhat more futuristic outlook they could have several different sensors measuring the object that is being modified. The data of the sensors could be visualized to various screens. At the same time the worker could have a pair of augmented reality glasses which could show some other aspect of the status and the progress of the object. In addition, a reasonable number of external displays could be near the worker and each of them could show either further instructions or different status messages and they could even have machine learning forecasts about the object and how the work is progressing.

In this futuristic scenario the development of the applications of the devices would greatly benefit from the liquid software philosophy. The liquid software approach makes sure that the data and state synchronization is thoroughly thought out. If one device is then replaced with a completely new one, it would not affect the whole system as there would likely be a certain level of abstraction between the devices. Without liquidity the worst scenario would be that the logic between necessary devices would have to be implemented from scratch. Again, this case will not require full liquidity if the real-time requirements of different devices and their data synchronization is not an issue; nevertheless the liquid philosophy approach could bring up important questions about the whole system and its requirements.

2.3.3 Collaboration scenario

In a collaboration scenario multiple users can use several devices that are constantly synchronized. The collaboration scenario can be either sequential or simultaneous. One addition the collaboration could bring to the sequential and simultaneous screening scenarios described before is the monitoring of each task. This could be used to improve task schedules or prepare for delays or disasters. On a work site the workers themselves could see the results of their own tasks and also see the results of nearby tasks of other workers from which they could already tell what the next task is going to be and what equipment or initial data is required to complete it. The monitoring in a collaboration scenario should not be confused with micromanaging.

The collaboration scenario is especially important in a global world where a work project could have people from various time zones. When the liquid design philosophy is taken into account, the data movement between each project member should be hassle free and the restrictions to get the latest data should be minimal.

2.3.4 Overall benefits

The features of liquid software are not particularly new but they cannot be taken for granted either. I will now list some benefits of satisfying each requirement of the requirement list of 2.1.1. Satisfying the 1st and 2nd requirement means that the user experience is polished and moving from one device to another does not require additional work by the user which can save a large amount of time. Satisfying the 3rd requirement means that data is always up to date and even if the user was to move from an offline to an online state the data would still be synchronized and only data conflicts would require more effort from the user. The 4th requirement makes it possible for the user to continue their work without any waiting or downtime even if they are constantly moving between the devices. The 5th requirement gives the user the freedom to choose their own tools without having to worry about vendor lock-in. Finally, satisfying the 6th requirement gives the user the ability to modify the tools to their own needs since different tasks have different requirements and different people have different ways of working.

3. TECHNOLOGIES

3.1 Basic concepts of client-server model

In [48] there is a discussion about what is data and what is information and why they are important. Companies are highly dependent on data since it enables them to do business. In order to get information we need to process, organize and structure data and in order to preserve and to pass information on we need to store data.

The evolution of data storing and databases is briefly addressed in [15]. Data was first stored only on the applications but later it evolved in to a client-server architecture where all persistent data was kept on a database server and the application logic was mainly handled by the client side. The client would fetch the persistent data from the server and then process it and show it to the user. Another look at client-server architecture is in [9] where it is said that "In a client-server architecture, your program is broken up into two different pieces that typically run on two separate computers. A server does most of the heavy lifting and computation; it provides services to its clients across a high-bandwidth network." One major benefit of client-server architecture is presented in [27]; systems with different processor architecture, different operating systems and different libraries can co-operate via the servers.

Liquid software usually uses client-server architecture because it helps different devices with different environments to work together since all the data is passed through the server. The use of a server helps in moving the logic to one place so it can be implemented only once and when the functionality is on the server it is also closer to the data. Usually it is not enough to store the persistent data solely on the server and especially in the case of liquid software it must be possible to store persistent data on the client side. The connection to the server could be lost which would render the client useless without local data storage. Different devices can also use the same data in different ways so instead of implementing the data handling of

each device to the server side it is better to share the computing load between different parties and implement the client side data handling for each device. This also makes the solution more modular which reduces the dependency between different clients and servers.

Since the persistent data is stored in the client it would also be helpful to modify the data without being constantly connected to the server. This feature raises another issue: data synchronization. Because the persistent data can be in a different state on the client and on the server it is necessary to synchronize the data between the two parties. One important part in [15] regarding data synchronization is the concept of ACID transactions:

- "Atomic: The transaction is indivisible: either all the statements in the transaction are applied to the database or none are."
- "Consistent: The database remains in a consistent state before and after transaction execution."
- "Isolated: While multiple transactions can be executed by one or more users simultaneously, one transaction should not see the effects of other in-progress transactions."
- "Durable: Once a transaction is saved to the database (in SQL databases via the COMMIT command), its changes are expected to persist even if there is a failure of operating system or hardware."

These requirements are necessary if the integrity of data must be sustained.

Another interesting aspect is the popularity of RDBMS and SQL as a way to store persistent data in the server and the popularity of OOP as a way to create program logic. As [15] tells, converting data from one form to another is not a simple task but it is often required when transferring data between an application and a database. This is also important when we focus on the ways the browser stores data locally.

Data update conflict is also an important issue. Certain conflicts can be handled by satisfying the ACID requirements but some conflicts require more logic and possibly even human interaction in order to achieve the desired outcome. Conflicts would occur when two parties are modifying the same resource at the same time. They

might fetch the same data from the server and modify it. When both parties upload their own modifications to the server, one of the modifications could be lost. Different conflict detection and handling implementations exist but they are not in the scope of this thesis. Instead it is enough to understand the possible use cases where a conflict might happen. In a liquid environment conflict handling is very important as the user can switch between offline and online state all the time and the data is often shared between different users and devices.

3.2 Backend as a Service (BaaS)

One good architecture for liquid software is the client-server architecture because it helps to keep the latest persistent data in one place and it helps to share the computing load between the client and the server. The difficult part about client-server architecture is the implementation of the server because it often needs to be scalable and be prepared to serve a large amount of clients. There are some options for the developer to buy the server as a service so they do not need to implement everything themselves.

Backend as a Service, in short BaaS and sometimes referred to as Mobile Backend as a Service, is a service model that provides mobile application developers with a scalable cloud backend which has a set of important features that the mobile applications usually require. The biggest benefit of using a BaaS is that the developer does not need to build their server from scratch because BaaS provides a server where the low level configurations have already been made. It aims to improve from Infrastructure as a Service and Platform as a Service models by giving the developers more features and an easier way to build their server for the mobile applications [47]. The key features of BaaS providers are usually push notifications, user management, data storage, file storage, support for offline use and easy construction of API's between the client and the server [53]. BaaS implementations can also provide SDKs or software libraries for the client which make the use of BaaS related services easier.

In addition to providing an almost ready server the BaaS has important features which are often required by liquid software applications. Some of the key features are: user management, push notification, data and file storage and offline support. If the software has more than one user, a user management system must be implemented into the server in order to distinguish between different users and their

devices. The push notification feature is optional but if it is missing, a bidirectional communication channel between the client and the server or a frequent data polling feature must be implemented in order to keep the devices synchronized. Data storage in the server is a mandatory feature because the clients often can not store large amounts of data. The data storage on the client is mandatory because offline usage requires it. The support for offline usage is also mandatory for the application to be considered truly liquid because if the support is not provided then the roaming between devices is no longer hassle-free nor fluent in case the connection to the server is lost. File storage is mandatory for the server but on the client it can be either optional or mandatory based on the way the client uses the files. If the developer does not want to use an existing BaaS or another service which provides similar features then they should be prepared to develop those features themselves. Example BaaS systems that are listed in [53] are Kumulos [24], Kinvey [22], AnyPresence [3], and Kii [21] and the features they provide can be found from their websites.

Using a BaaS may introduce a major problem: vendor lock-in. If enough of the features of the BaaS provider are used, it might be difficult to transfer to a different provider if the logic behind some features is different or if the features are used through a specific library that is implemented by the provider.

3.3 Browser storage technologies

Because the point of this thesis was to study how a liquid software could be made for a web browser it would be good to take a look at some data storage technologies of the browser. The data storage capabilities of the browser are important because a liquid software application is expected to have support for offline usage. If the web application loses the connection to the Internet then the data needs to be stored locally or else the data would be lost once the application shuts down. The amount and type of persistent data that must be stored sets some requirements for the data storage of the browser.

The browser support and data size restrictions vary between the technologies and they all have their own use cases. All of these technologies are bound to a single origin and they implement the same origin policy [61] which means that they can not be modified from anywhere else but from the origin where they were created.

LocalStorage [65] is a simple key-value pair storage which is easy to use and which

has a large browser support. It also has a very limited storage size which makes it inadequate to store large amounts of data. The keys and values are stored as strings so the LocalStorage does not have good support to store complex data types. It is a very good data storage for simple data types when the required amount of storage is low.

IndexedDB [62] is a transactional object-oriented database. It can store all data types that are supported by the Structured clone algorithm [34]. Based on the web browser the IndexedDB either has no limit to the size of the storage or it has a size limit which is based on the quota of the browser. The quota size is usually some percentage of the total disk space of the device it runs on. IndexedDB is a good data storage when large amounts of data needs to be stored or if the stored data type is complex.

Service worker [64] is a background worker for the web browser. The service worker is registered for a certain web page and after that it continues to work inside the browser even if the web page is closed as long as the browser stays open. Service worker contains a resource cache which can be used to store web resources that are usually fetched from the server every time a web page is loaded. It is a good storage when the user requires offline usage support for the resources of the web application.

3.3.1 LocalStorage

LocalStorage is a data storage of key-value pairs and it runs on a browser and it is part of Web storage which is also known as DOM Storage. Both the key and the value are stored as strings. Its specification can be found from [65]. It was developed to replace cookies in data storing and to give a way to store data that spans across multiple windows and to store data that lasts even after the session has ended.

The concept of origin was discussed in section 3.3. The LocalStorage is bound to a single origin. The web client is able to limit the size of the storage quota and for each origin a limit of five megabytes is suggested. The final storage limit depends on the browser. The browser support for LocalStorage is very good so using it will not affect the supported browsers. Wide support for web browsers is important for a liquid application because then the user can decide which browser they want to use. This would help with the requirement called “Freedom of ecosystems” that was presented in the requirement list from the liquid software manifesto in 2.1.1.

LocalStorage is not the ideal storage for liquid software solutions because of its small size quota and the limitation of data types to strings. It preserves the persistent data of the storage between shutting down and restarting the browser but the data can not be modified and accessed unless the user has a web page open for the application. This means that LocalStorage can not be synchronized in the background. Fortunately there are other ways to store data in a browser.

3.3.2 IndexedDB

LocalStorage is fit to store persistent data in use cases where the size requirements are low and the data is not very complex. In many cases there might be a need for a bigger data storage that supports more complex data types. This is where the IndexedDB comes in.

According to [35] the Indexed Database, in short IndexedDB, is a transactional database system for the browser and it has a low-level API for client-side modifications. It can hold significant amounts of structured data of all types that are supported by the Structured clone algorithm [34] which has support for various primitive types, JavaScript objects with cyclical structures and for files and blobs. The size quota of IndexedDB is much larger than the quota of Web storage. The browser can contain several indexed database instances and they all are bound to some origin [62]. In an origin each database must have a unique name which stays constant for the lifetime of the database. Each database also has a version number which starts from 0.

Even though IndexedDB is a transactional database system it is not completely similar to SQL-based RDBMS [35]. SQL-based RDBMS use fixed-column tables while IndexedDB is a JavaScript-based object-oriented database. In IndexedDB everything is stored as key-value pairs. The key is valid if it is one of the following ECMAScript types: Number, String, Date or Array [62]. The data of IndexedDB is modified and fetched with transactions and they are always done asynchronously.

According to [62] each database consists of object stores which are the primary storage mechanism for storing data in a database. Each object store has a name to identify it within the database. The data is stored in an object store as a list of records which consist of a key and a value. The list is sorted according to key in ascending order and the keys are unique which means that two records can not have

the same key. Keys can be created in the following ways: with a key generator, via a key path or by explicitly specifying it when a value is stored.

Like the name implies the IndexedDB can use indexes for searching items from the stored data. An index allows one to look up records in an object store based on the properties of the value of each key-value pair [62]. An index is always related to an object store. The records in an index are automatically populated whenever records in the referenced object store are inserted, updated or deleted. Each object store has multiple indexes which means that if the object store is modified then all of its indexes are modified as well. The records in an index are always sorted according to the key of the record.

Whenever data is read or written to the database it is done by using a transaction. According to [62] they represent an atomic and a durable set of data access and data mutation operations. Transactions are isolated from each other which means they can not modify the database simultaneously. The level of isolation is based on the mode of the transaction which can be either read only or read-write. The specification does not mention anything about consistency though so the transactions can not be said to be completely ACID based on the specification. The transaction has a fixed scope through its lifetime that determines the object stores which the transaction may use.

IndexedDB has support for cursors. They are a transient mechanism used to iterate over multiple records in a database. The storage operations of a cursor are performed on the underlying index or an object store. [62]

Since the IndexedDB stores key-value pairs it will cause an object-relational impedance mismatch if the server uses SQL-based RDBMS and the data needs to be stored from server to browser. This means that the data needs to be modified and restructured when moving it from IndexedDB to an SQL database and vice versa. Such use case is necessary when the application needs to be used in offline mode or if the use of bandwidth should be reduced.

The best part about IndexedDB considering liquid software is the fact that in can be used by a service worker which will be covered in 3.3.3. This makes it possible to modify the database in the background even if the user would not have the web page open as long as the browser is running and the site has been visited at least once.

3.3.3 Service Worker

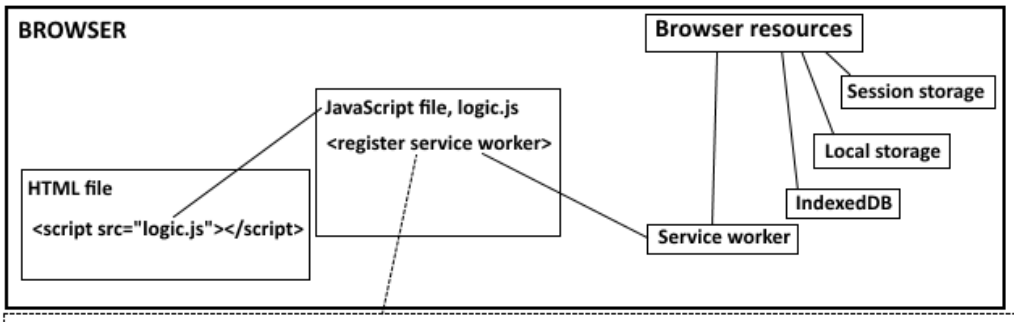
IndexedDB and LocalStorage are good ways to store the persistent data of a liquid software application. In order to fully support the application in offline state the UI resources need to be stored somewhere on the client. Service worker has a solution for this.

In a browser a web worker [67] is a worker instance that is spawned by the main thread. The worker can run scripts in a parallel thread. A service worker [64] is an event-driven web worker that has some additional features. It executes on the web browser and it is created by writing a JavaScript file that implements the service worker interface. It can be added to be part of a web application by registering it against the origin and the path of the application [36]. If registration is successful then the file containing the service worker implementation will be downloaded to the client where it will be installed and/or activated. From the outside the service worker seems like any normal JavaScript file but the main difference between them is that the service worker must be registered and then installed and finally activated by the browser.

In figure 3.1 is explained how service worker is related to a web application or to other browser resources. In the image the HTML file and the JavaScript file could together form a simple web application. The JavaScript file has a function which registers the service worker. In order to access the service worker in a web application it must be called through the Navigator [46]. The Navigator returns a **ServiceWorkerContainer** object [38] which contains the **register** method. The register method returns a **ServiceWorkerRegistration** object [39] when called and from that object the currently active service worker instance can be fetched. This instance implements the **ServiceWorker** interface [37] which is derived from **Worker** interface [60] and which both contain several methods that can be called from the web application.

A service worker is run in a worker context; therefore it has no DOM access and it runs on a different thread than the main thread that powers the web application. The service worker does not block the execution of the main thread [36].

All calls to the service worker are executed asynchronously which means that it can not use any of the synchronous APIs provided by the browser. However it can use the asynchronous APIs of which IndexedDB is the most important one regarding




```
function registerServiceWorker() {
  if ('serviceWorker' in navigator) {
    navigator.serviceWorker.addEventListener('message', function(event) {
        ChromeSamples.setStatus(event.data);
    });
    navigator.serviceWorker.register('service-worker.js').then(function(registration) {
      Subscribe(registration);
    });
  }
  else {
    alert('Service workers aren\'t supported in this browser.');
  }
}
```

Figure 3.1 *A diagram explaining the relationship between a service worker and other browser modules.*

liquid software and this thesis as it can fetch data from the database and modify its contents on the background.

According to [40] when the service worker is installed it is always installed in a certain scope. A scope is nearly the same thing as the origin of the web site which was discussed in section 3.3. The difference is that the scope can be made more precise by giving it some URI that resides under the origin. The service worker also has a cache for the resources of the web site such as HTML documents and images. The developer can tell the service worker the names of the resources that should be in the cache. The service worker also has a method called 'fetch' which can be used to get resources from the same scope where the service worker is installed. Each time resources are loaded the service worker can first check the cache and return the cached version instead. If the resource does not exist in the cache then it can be fetched from the server and cached. If the resource can not be obtained from the server then a fallback resource can still be sent to the user.

The cache used by the service worker is designed to replace Application Cache [54] which is a set of cached resources that is supposed to help the developer to create offline web applications. According to WHATWG in [54] it is currently in the process

of being removed from the Web platform. WHATWG instructs to use service workers instead if offline features need to be implemented to a web application.

Another important feature of the service worker is its ability to receive push notifications which means that the service worker can synchronize data with the server completely in the background without disturbing the main thread. When the user finally returns to the application then all the necessary changes to the data could already be synchronized and up to date. This is done with the help of Push API which will be covered in 3.4.4.

Service workers are started and kept alive by their relationship to events instead of documents which means that a service worker instance may be started by the web browser to execute some task and after the task is done the service worker is killed by the browser [64]. The service worker would have done all this without ever receiving a message from the document it is installed to. Even though the user closes the web page the service worker may stay alive as long as the browser stays on. Even if the browser kills the service worker instance it may start the service worker again when needed.

The lifecycle of a service worker works as follows: It is first registered, then downloaded, then installed and finally activated [36]. If an older version of the service worker already exists then the new worker will wait until all the web pages using the old one are closed. After that the new worker can activate and start executing.

Two important technologies are closely related to the service worker: message channel and web notification. They both improve the use of service worker and they will be covered next.

Message Channel

The Channel Messaging API allows two separate scripts running in different browsing contexts attached to the same document to communicate with each other [33]. With this API the service worker can communicate with the main JavaScript thread which in turn can modify the DOM to show the latest data to the user. This way it might be enough to implement the IndexedDB functionality to just service worker and transfer the various commands and data through the message channel which could simplify the architecture of a web application.

Web notification

If a user needs to know when the persistent data has been synchronized to the server or that the data has changed while using the application the browser can show the user a notification of such action with the Notifications API [41]. The API lets the browser pop up a notification on the web page and the notification is displayed outside the page at a system level. This means that the user may receive notifications even if the application is idle or in the background. According to [58] the notification API is designed to be compatible with existing notification systems while remaining platform-independent.

Notifications have several attributes such as title, body, data etc. and they can either be non-persistent or persistent. If a notification is associated with a service worker registration it is considered to be persistent and in the opposite case it is considered to be non-persistent. The difference between persistent and non-persistent notifications is that the persistent one can have actions which can be shown to the user in the notification window so the user can choose which action to take when the notification pops up. The persistent notification would then be handled in the service worker which in turn would choose the task to execute based on the action that was chosen by the user.

Notifications require the permission of the user in order to be displayed. By default the notifications are denied and when the user navigates to a web page with notifications they are first asked if they wish to allow the notifications from the web origin of the web page. Based on the implementation of the browser the notification should be shown in their own display area apart from the active DOM of the web page.

The tag attribute of the notifications makes it possible to update older notifications or to make sure that if the user has several instances of the same web application open then only one notification is received. If the user receives multiple notifications with the same tag then they are seen as one notification where the latest one overrides the older ones.

3.4 Browser communication technologies

Liquid application may have some requirements about the communication channel that is used between a server and a client. To satisfy the "Effortless roaming" and "Usability" requirements of liquid software from 2.1.1 the communication channel should be bidirectional and full-duplex which means that both parties can communicate with each other simultaneously.

The browser has different ways to communicate with the server. XmlHttpRequest [57] is the oldest one and is supported by all browsers but with the evolution of the web it has been noticed that simple document requesting is no longer enough for the web sites to provide a good functionality for their users. Different technologies have emerged with the rise of HTML5.

WebSocket [16] is a bidirectional communication channel between the server and the client. With WebSocket the client and the server establish a TCP/IP channel between them and they can both send data to the other party at any time. It has low overhead and

Server-sent events [55] is a communication technology where the client and server establish a connection between each other and after that the server can send data to the client. The client can not send data through the channel. The data type that can be sent is restricted to UTF-8 encoded strings.

Web push [18] or Push API [63] is an upcoming communication technology that is similar to server-sent events in the sense that with it the server can send data to the client but the client can not send data to the server. The difference is that in server-sent events there is a constant connection between the server and the client but in web push the client is connected to a push service which can receive XmlHttpRequests from the server and then send the data to the client.

3.4.1 XmlHttpRequest

According to the XmlHttpRequest living standard by WHATWG the "XmlHttpRequest defines an API that a provides scripted client with functionality for transferring data between a client and a server" [57]. The client starts by creating an XmlHttpRequest object which implements an interface that contains event handlers

and a state property. When the XmlHttpRequest object is created the client sets properties such as the URL of the server and the type of request to the object. The request is then sent to the server and then the event handlers of the XmlHttpRequest object begin notifying the client on the changes to the state of the object. When the request is handled or fails the state is set to **done** and then the client can read the response of the request.

XmlHttpRequests can be asynchronous or synchronous but according to the living standard the synchronous feature is on the process of being removed from the web platfrom. The data that can be sent and received through an XmlHttpRequest can be ArrayBuffer, Blob, HTML document or JSON.

XmlHttpRequests are good for fetching different resources and persistent data for the client but the problem with them is that in order to get constantly updated data from the server the client needs to poll it repeatedly with new requests. The polling has a large overhead and causes unnecessary request handling for the server. Luckily there are other ways to implement communication between the client and the server.

3.4.2 WebSocket

The speed of persistent data synchronization might be essential for a liquid software application if the amount of synchronized data is constantly very high. With the HTML5 a new communication standard was introduced which was designed to enable continuous communication between a client and a server.

The WebSocket Standard [16] says that "The WebSocket Protocol enables two-way communication between a client and a remote host that has opted-in to communicate with the client". This means that the server and the client can send data to each other through the WebSocket channel once the connection has been established. Certain web applications require a bidirectional communication channel between the client and the server. Before WebSocket standard the bidirectional communication was implemented by abusing the HTTP polling which resulted in a variety of problems. One such method that abuses the HTTP is called long polling where a client sends a request to a server and the request is kept alive by the server until it has some new data to send and once the client receives the data it sends another request which is again kept alive by the server until new data can be sent. The WebSocket

Protocol was designed to fix this.

According to [16] the communication channel between the client and the server is created by establishing a TCP/IP connection between the two parties. Before the TCP/IP connection can be established a handshake is performed over HTTP traffic; after that the communication can begin. WebSocket was designed to have a minimal overlay for each message. This way it is ideal for bidirectional, full duplex communication and it has good support for the use case where the frequency of messages between the client and the server is high. Web sockets use the same origin policy which means they will deny messages from a source that does not belong to the same origin as the WebSocket object. WebSocket can use both unencrypted and encrypted connections.

In [56] is the standard for the WebSocket API and in it is stated that WebSocket uses events to inform the user application of its state. The API has event handlers for opening the connection, closing the connection, receiving a message and for error messages. The data types that can be sent over WebSocket are strings, Blob objects and ArrayBuffer objects.

The WebSocket Protocol standard [16] states that the only relationship the WebSocket Protocol has with HTTP is that the handshake event is interpreted by HTTP servers as an Upgrade request. An article by P. Lubbers [26] tells that a WebSocket connection is established by upgrading from the HTTP protocol to the WebSocket protocol. The article also says that there often might be proxy servers between the client and the main server that is providing the web page content and this can be a problem for the WebSocket since the connection is supposed to be working until the client or the server explicitly wants to disconnect. The proxy however might not understand the WebSocket protocol and all it sees is the use of the HTTP protocol in the handshake phase but after that the proxy might close the connection because it thinks that either one of the parties has become unresponsive. Liquid applications require a hassle-free and effortless roaming so possible connection problems with proxy servers should be taken into account when developing a liquid application.

3.4.3 Server-Sent Events

Sometimes it is enough for a liquid application to simply send requests to the server through the XmlHttpRequest protocol without having to use the constant connec-

tion of the WebSocket. Nevertheless it would be useful if the server could send data to the client without the client having to poll for it every few seconds. This is where the server-sent events come along.

Server-sent events are described in the HTML standard [55]. They are used through an EventSource interface which is described in the part '9.2.2' of the HTML standard. They enable the server to push data to web pages over HTTP so the client does not need to poll the server for the latest data. The server-sent events are constructed by creating a consistent connection between the client and the server so it is similar to WebSockets. The differences to WebSockets are that the server-sent events have an automatic reconnection functionality and that the client can not send data to the server through the EventSource interface. The server-sent events use only HTTP while the WebSocket uses an upgraded version of HTTP protocol to communicate.

The data of server-sent events contains the **text\event-stream** MIME type. [55] says that the Event streams are always decoded as UTF-8 and that there is no way to specify another character encoding. Sending data therefore is heavier than it is with web sockets.

If the connection between the client and the server is lost then the EventSource automatically tries to reconnect. It has a **readyState** attribute which tells whether it is connecting, open or closed. It also has event handlers for opening a connection, receiving a message and for error messages.

The biggest drawback for EventSource is that it is not supported in any Internet Explorer or Microsoft Edge [29] browser versions. It is still in a draft state and the W3C is no longer working on it. Some other drawbacks are acknowledged in the '9.2.6 Authoring notes' of [55] such as that legacy proxy servers might drop long HTTP connections such as the EventSource connection but this can be prevented by sending extra messages every 15 seconds to keep the connection alive. Another acknowledgment is that HTTP chunking might cause unexpected behavior on the EventSource protocol.

3.4.4 Push API/Web Push

WebSocket and server-sent events can be used to synchronize persistent data between a client and a server. They both can also communicate with a service worker

which was discussed in 3.3.3. Still the IETF and W3C are working on a new communication technology which would allow the server to push data to the client. The new technology is known as Push API [63] or web push [18] and it is designed to support the new HTTP/2 protocol [17] which is supposed to improve the performance of the HTTP. An important aspect in HTTP/2 is that it is designed to use a single connection between a web page and a server rather than having multiple different connections. HTTP/2 will not be discussed further in this thesis.

Web push is a new communication protocol that is still in a drafting state. The protocol [18] is being developed by IETF and the Push API for it that also known as W3C Web Push API [63] is being developed by W3C.

The idea behind web push is to provide push notifications for web applications. Push notifications are messages that are sent from a server to the client so in principle they are the same as server-sent events but the push notifications have a different approach to them. According to [18] the general model for push services includes three basic actors: a user agent (the client), a push service (between the client and the server) and an application (the server). The user agent subscribes to the push service and begins monitoring the service for new messages. After subscribing to the service the user agent sends a push resource to the server. The server can now send a message to the push service based on the information of the push resource that was received from the client. When push service receives a message it forwards it to the user agent based on the information in the message.

Push API and the service worker were designed to work together since only the service worker has the ability to listen to the messages that are sent by the push service. All messages are attached with a property called 'endpoint'. It is an URI which specifies where the message should be sent. When a push service receives the message it forwards it to the correct client based on the endpoint. The message is then caught by the service worker that has been subscribed to the push service. The push subscription process can be seen in figure 3.2.

The client can unsubscribe from the push service when they want. When this happens the server should also remove the subscription endpoint from its memory and database. Subscription endpoints can change and when this happens a subscription update action should be made by both the server and the client. Each service worker can have only one subscription and each subscription is associated with one service worker.

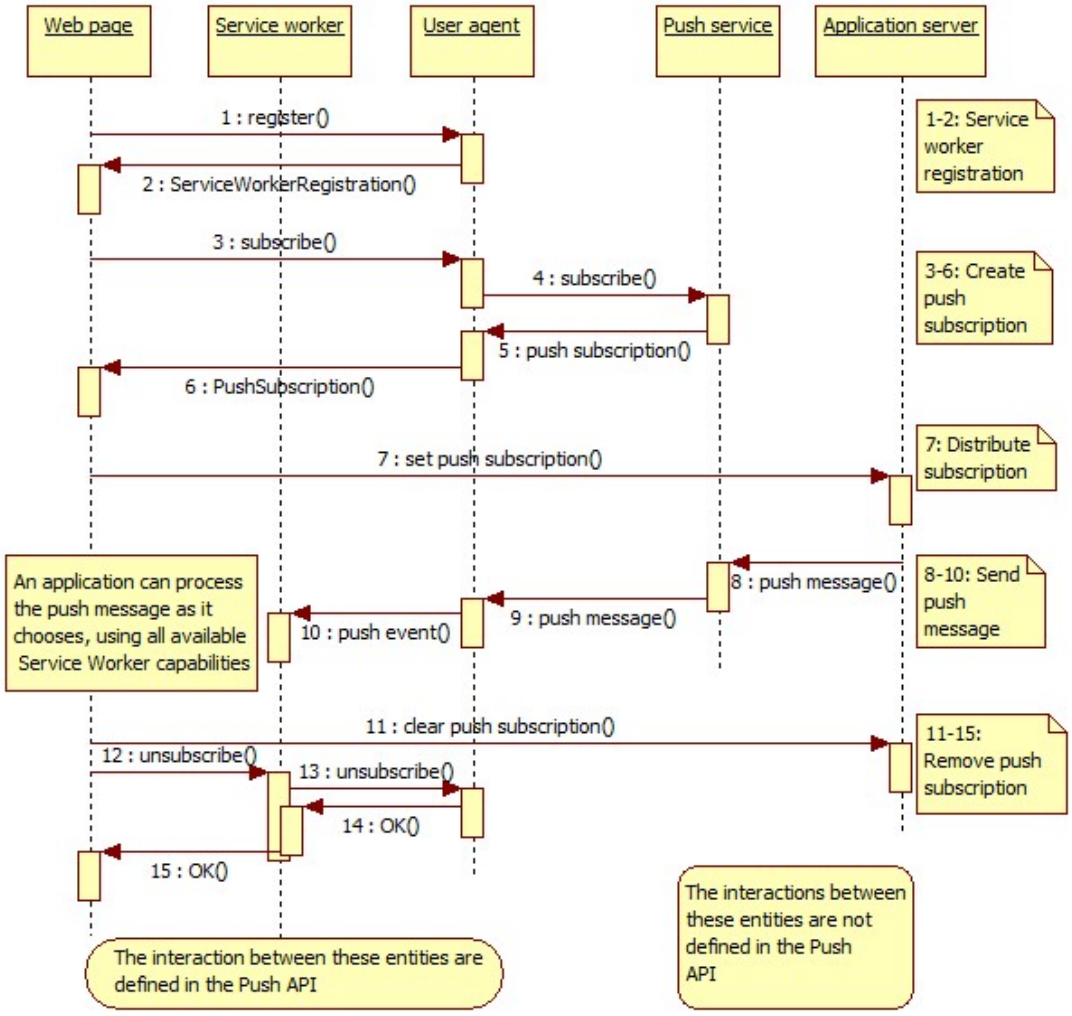


Figure 3.2 *Push subscription process according to a diagram in [63].*

According to both the API [63] and the technology [18] standards the use of an encrypted communication channel is mandatory for Push API communication. If the user tries to subscribe over unencrypted HTTP then an exception should be thrown. Of course the browser designers can decide how they want to implement the standard for the browser.

Push API technology has a very limited browser support and in the scope of this thesis it was tested only with Google Chrome. The implementation of Chrome currently has a limitation which prevents push service from sending any data in the push messages so they can only act as notifications for the client to start a data fetching process.

Chrome requires that the push service is implemented as a Google App Engine project [14]. Google App Engine is a platform as a service by Google. It provides developers with a server platform on which the user can host applications which can use the various features of the Google App Engine. To use Push API it is enough to create an empty Google App Engine project and then give the private key of the project for the main server of the liquid software application. When the server sends Google App Engine a push notification along with the endpoint and the key the Google App Engine will take care of forwarding the message to the client.

3.5 Conclusions

In tables 3.3 and 3.4 can be seen the storage limitations of different storage technologies. Quota means that the browser can use some percentage of the storage of the device. No limit means that the hard disk of the device limits the storage. RAM limit means that the data is stored in RAM and therefore the amount of RAM limits the amount of data that can be stored. The LocalStorage has very limited quota on all browsers which does not make it an ideal storage for liquid software. The amount of data that needs to be stored on the client can be very large because in offline mode the software has to function properly. WebSQL and Application Cache are both considered obsolete and FileSystem is only for Chrome. Application Cache has the best support among browsers. IndexedDB has fairly good browser support among the latest browser versions but service worker is still lagging behind. I would still choose both IndexedDB and service worker for data storage in a liquid web application. Service worker is mandatory in order to get the most liquid-like features because it is the only one that gives support for background synchronization.

In table 3.1 a list of browsers and different browser storage technologies can be seen. In each row is the earliest version number of the browser that has support for the technology. If the browser does not have any versions which completely support the technology then a version that supports it partially has been selected. Such a version can be identified from the parenthesis surrounding the version number. If the technology does not have even partial support then a hyphen is in the place of the version number. A similar table for communication technologies can be found in table 3.2. All the data has been collected from [8]. Note that XmlHttpRequest is implemented in all browsers but the advanced features are not supported in IE. The (Adv) after XmlHttpRequest refers to advanced. Also note that SSE refers to Server-Sent Events.

Browser	LocalStorage	IndexedDB	Service Worker	AppCache
IE	8	(10)	-	10
Edge	12	(12)	-	12
Firefox	3.5	10	(44)	3.5
Chrome	4	23	(40)	4
Safari	4	(7.1)	-	4
Opera	11.5	15	(24)	11.5
iOS Safari	3.2	(8)	-	3.2
Android Browser	2.1	4.4	(47)	2.1
Opera Mobile	12	36	(36)	12
Chrome for Android	49	49	(49)	49

Table 3.1 *Data storage support in browsers. Table data is from [8].*

Browser	XmlHttpRequest (adv.)	WebSocket	SSE	Push API
IE	(10)	10	-	-
Edge	12	12	-	-
Firefox	12	11	6	44
Chrome	31	16	6	(44)
Safari	7.1	7	5	-
Opera	18	12.1	11.5	-
iOS Safari	8	6.1	4.1	-
Android Browser	4.4.4	4.4	4.4	-
Opera Mobile	12	12.1	12	-
Chrome for Android	49	49	49	(49)

Table 3.2 *Communication support in browsers. Table data is from [8].*

As for communication technologies I think they all are sufficient for developing a liquid web application. All of the technologies of section 3.4 can be used in a service worker at least on Google Chrome as long as the communication goes over HTTPS protocol and it is asynchronous. The communication from client to server could be done with XmlHttpRequests. Push API would be the best choice for sending data from server to client as it is designed to work together with the service worker and the concept of a push service reduces the need for active connections on the main server. A proxy server for WebSocket and the server-sent events type communication could also be made between the web application and the main server. However, the server-sent events is no longer developed and the forced UTF-8 encoding limits the data types. WebSocket itself is a very different protocol and designed for specific use cases so if the data synchronization was done over a REST interface then WebSocket would not be the ideal protocol to use.

	Chrome 40	Firefox 34	Safari 6, 7	Safari 8	IE 9	IE 10, 11
Application Cache	Up to quota	No limit	No limit?	No limit?		100MB?
FileSystem	Up to quota					
IndexedDB	Up to quota	No limit		Up to quota?		10MB, 250MB, (999MB)
WebSQL	Up to quota		5MB, 10MB, 50MB, 100MB, 200MB..	5MB, 10MB, 50MB, 100MB, 200MB..		
Local Storage	10MB	10MB	5MB	5MB	10MB	10MB
Session Storage	10MB	10MB	RAM limit	RAM limit	10MB	10MB

Table 3.3 *Browser storage limits of desktop browsers. A table from [23]. Data is from 2014.*

	Chrome 40	Android Browser 4.3	Firefox 34	Safari 6, 7	Safari 8	iOS WebView 6, 7	iOS WebView 8
Application Cache	Up to quota	No limit?	No limit	300MB?	300MB?	100MB?	100MB?
FileSystem	Up to quota						
IndexedDB	Up to quota		No limit		Up to quota?		Up to quota?
WebSQL	Up to quota	200MB		5MB, 10MB, 25MB, 50MB	5MB, 10MB, 25MB, 50MB	50MB	50MB
Local Storage	10MB	2MB	10MB	5MB	5MB	5MB	5MB
Session Storage	10MB	RAM limit	10MB	5MB	RAM limit	5MB	RAM limit

Table 3.4 *Browser storage limits of mobile browsers. A table from [23]. Data is from 2014.*

4. PRINCIPLES OF THE SOLUTION

In this chapter I will go through the principles of a solution that would satisfy the various requirements set by the liquid software manifesto and this thesis. The solution must comply with the restrictions that are set by the web environment.

4.1 Overall description

The proof of concept solution must run on a web browser and in this solution the browser is limited to Google Chrome. A server-client architecture would be a good choice because the client can be kept simpler and the persistent data can be kept on the server. Synchronization of the persistent data between the server and the client is easier than it would be in a peer-to-peer architecture because in the server-client architecture the final data is always in one place.

XmlHttpRequest can be used for basic communication between the client and the server. With it, the client may fetch resources and persistent data from the server and send data modification requests but the server can not use the XmlHttpRequest to push persistent data to the client. In liquid software it is important that the synchronization of the persistent data is kept fast. The client could continuously poll the server for data changes but if the unnecessary polling should be avoided then the Push API could be used to implement a communication channel from the server to the client. If Push API is used then a service worker must be implemented into the client side. Because the solution must work in Google Chrome an additional Google App Engine project must be set up to act as a push service for the Push API. This is a requirement of the Google Chrome browser.

The client has some user interface and logic both of which can be implemented on Google Chrome with HTML, CSS and JavaScript. The synchronization module requires a service worker in order to receive notifications through the Push API. So far the client may be interacted with, receive data update notifications and fetch

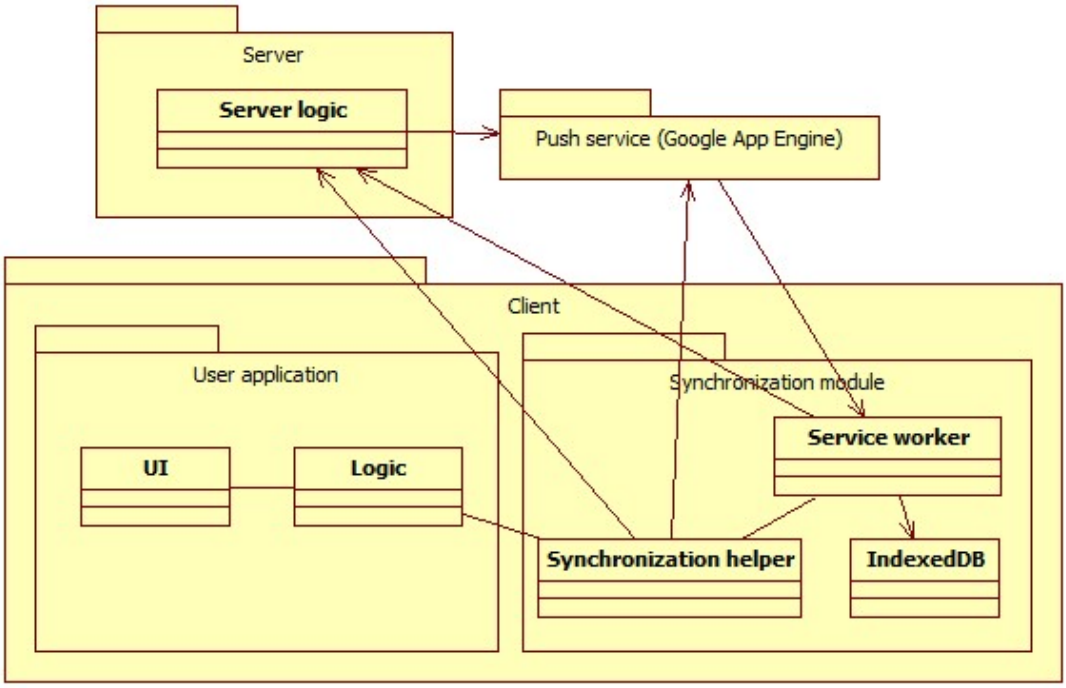


Figure 4.1 *Overall architecture of the solution*

new data from the server. If the client loses its connection to the server, the client could still recover and keep all of the local data in its memory as long as the browser is not shut down. If the user would still wish to navigate through the different views of the client, a cache for web resources is required. The service worker has a cache which can store web resources. It is now possible to implement an application which can be used offline as long as the browser stays on.

In order to support a use case where the browser may be closed at any time a client side database must be implemented. LocalStorage could fit this role but if the data is expected to exceed the 5 MB storage limit of LocalStorage or if the data is so complex it can not be stored as String-types then another solutions is required. IndexedDB can store a large amount of data and it supports multiple data types. Because it has an asynchronous API the service worker can use it. Since the service worker runs on its own thread, a background synchronization for the persistent data can be implemented by allowing the service worker to modify the IndexedDB.

The architecture of the final solution can be seen in figure 4.1. The synchronization helper is not mandatory but it would be a good practise to keep a level of

abstraction between the client side logic and the common persistent data storage and synchronization methods. If the client wishes to send data to the server it is done through the synchronization helper.

4.2 Requirements by the Liquid software manifesto

The requirements of a liquid software are listed in subsection 2.1.1. Each item in the following list matches to the requirement list with the number and the name. The following list contains information about what restrictions each requirement sets for the proof of concept application of this thesis.

1. Effortless roaming: The server supports concurrent connections from multiple devices. The UI of the application works on desktop and mobile devices.
2. Usability: Offline functionality will be made for this with offline status checking and by storing necessary resources on the client. The persistent data will be stored in an IndexedDB and the web resources will be stored in the resource cache of a service worker.
3. Data synchronization: Will be implemented for the persistent data that is related to the logic of the software and for the resources used by the client application.
4. State synchronization: Not in the scope of this project. Will not be implemented.
5. Freedom of ecosystem: The demo software will be implemented on Google Chrome only but since the implementation itself is based on browser technology it could be implemented to other browsers as well. The browsers must support the necessary W3C standards.
6. Control: Not related to the core part of this thesis. Will not be implemented. The level of synchronization shall be fixed at all times.

4.3 Functional specifications

The data synchronization module will implement persistent data synchronization which happens in the background even when the application is not open. The data

synchronization must work at all times as long as the browser is running and the connection to the server is working. Only the persistent data that is related to the resources of the client or related to the functionality of the application will be stored and synchronized. Any operating system or browser related information will not be synchronized or stored.

The synchronization module will offer an offline functionality which will help the user to store data even if the client loses the connection to the server. If the connection is lost, the data synchronization module should still save all the data modifications made by the client so they can be synchronized with the server when the connection recovers. All persistent data is stored in a data storage of the browser so if the client loses the connection to the server and the browser is closed the data can still be retrieved from the storage once the browser and the client are started again.

The module will not check the data structure and will not be interested in the data. However, if the data is to stored in the database and important keys regarding the data storing are missing then an exception will be thrown. All the persistent data that is important for the logic of the application is stored in a local database. All web resources such as images and HTML documents are stored in the cache of the service worker. The browser may provide additional restrictions to the use of the data storage technologies.

The synchronization module will use a REST interface and XmlHttpRequest to communicate with the server. The client must be able to receive the push messages sent by the push service. Push message receiving will be implemented based on the requirements of the underlying web browser.

Intelligent synchronization will not be implemented as it is not the core part of this thesis. The user application will decide what data it wants and any intelligence regarding the data synchronization will be very simple.

5. IMPLEMENTATION OF THE APPLICATION

The principles of a liquid software web application were presented in chapter 4. In this chapter the focus is on a proof of concept application that was implemented to evaluate the chosen principles and to research the chosen web technologies. The focus of the proof of concept application is on the synchronization of persistent data between a server and a web client.

5.1 Overall description

The proof of concept application was implemented as a web application which means that it can run on a browser. Even though different browsers exist it was decided that it is enough if the proof of concept application can run on Google Chrome. The overall solution consists of a Node.js [19] server, a Google App Engine push service and a client.

The client consists of two web applications and a data synchronization module. The web applications have separate user interfaces and logic modules. Both of them are linked to the data synchronization module which consists of a data synchronization helper called Lsyncr, a service worker and an IndexedDB instance. The data synchronization module handles the communication with the server.

If a user is using an application and wants to modify the persistent data of the application then the data modification request is sent to the server via the data synchronization helper. Once the server receives the request it stores the request data to its memory and sends notifications about the new data to a push service. All the clients have registered their service workers to the push service; when the push service receives a notification it will forward the notification to the correct service worker. When the service worker receives a notification it will query the server for all the latest data the server has to offer. The service worker will store the data it received from the server to the IndexedDB and notify the data synchronization helper

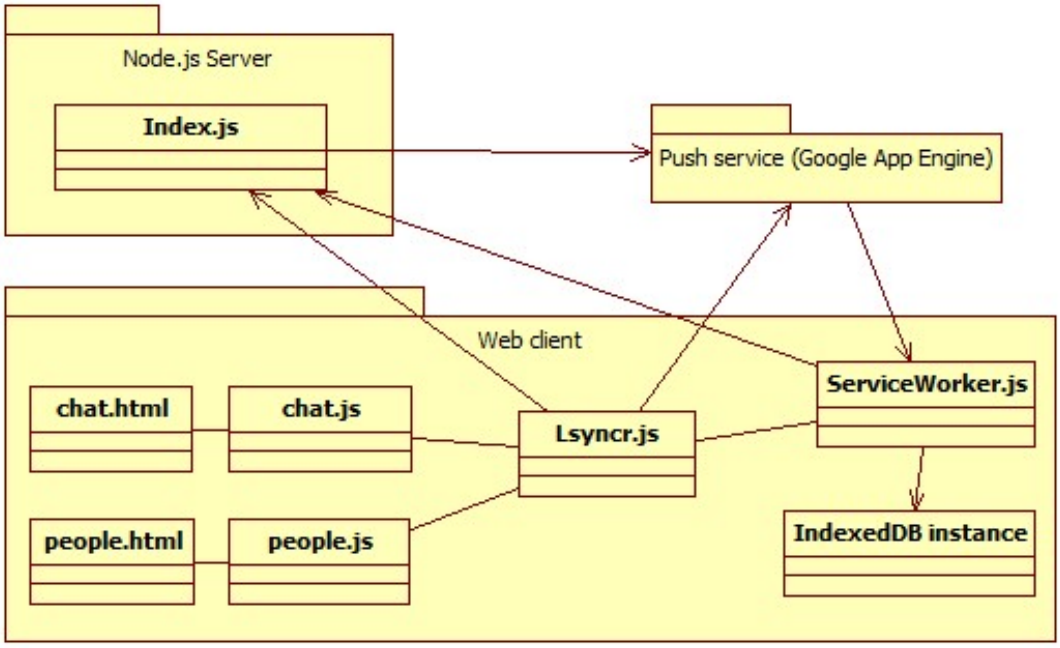


Figure 5.1 *Overall architecture of the implementation*

who in turn notifies the active application. A diagram of the overall architecture can be seen in figure 5.1.

5.2 User applications

The client has two web applications. The first application implements a simple web chat and the second one implements an application where the user can create **person** and **house** instances and link them together. Both of the applications consist of a single JavaScript file and an HTML file. They both use the jQuery [20] framework to interact with the DOM of the web page. The applications do not have access to the database of the browser or to the Node.js server. They both import the JavaScript file of Lsycnr. This way they get a reference to the Lsyncr object which handles the communication with the server and the database.

The chat application was made to demonstrate the persistent data synchronization. The user can choose their user name and send chat messages which are forwarded to all other users. All chat messages are stored to the IndexedDB.

The second application was made to demonstrate the persistent data storage and

```
$(document).ready(function() {
  var indexedDbName = "testapp";
  Lsyncr.setIndexedDbName(indexedDbName);
  Lsyncr.setPostCallback(addChatMessage);
  Lsyncr.setSyncCallback(addChatMessage);
  Lsyncr.setGetCallback(getChatMessages);
  Lsyncr.setReadyCallback(initialize);
  Lsyncr.start();
```

Figure 5.2 *Initialization and start of Lsyncr in a user application.*

the service worker cache. In it the user can create **person** and **house** instances. Each person and house is given a random name and an id when they are created. The user can link person and house instances together. Each house can contain multiple Person references and each Person can belong to one House. The user can freely move people in and out of houses. Additionally, each person is given an image id out of three possible choices; the server contains three different image resources with the same id values. On the user interface each person has their image next to their name and id. The references between houses and people made it possible to study how the circular reference affects the IndexedDB and the persistent data synchronization of the web application. The image files made it possible to study the service worker cache and how it works in offline mode.

5.3 Lsycnr and service worker

The synchronization library consists of three parts: the synchronization helper module called Lsyncr, the service worker and the indexed database. Lsyncr and the service worker will be covered in this section.

Lsyncr is implemented into a single JavaScript file. The Lsyncr instance is created when Lsyncr is imported in a user application file. The user application must initialize the Lsyncr instance before it can be used. An example of Lsyncr initialization from the point of view of a user application can be seen in figure 5.2. All the parameters except for the **indexedDbName** are functions. These functions will be called by Lsyncr once it has handled the specific request of the user application. The method given in **setSyncCallback** will be called each time the service worker of Lsyncr receives new data from the server. Lsyncr also has a method for setting the current user id. In the proof of concept application the user can choose from two possible user id values.

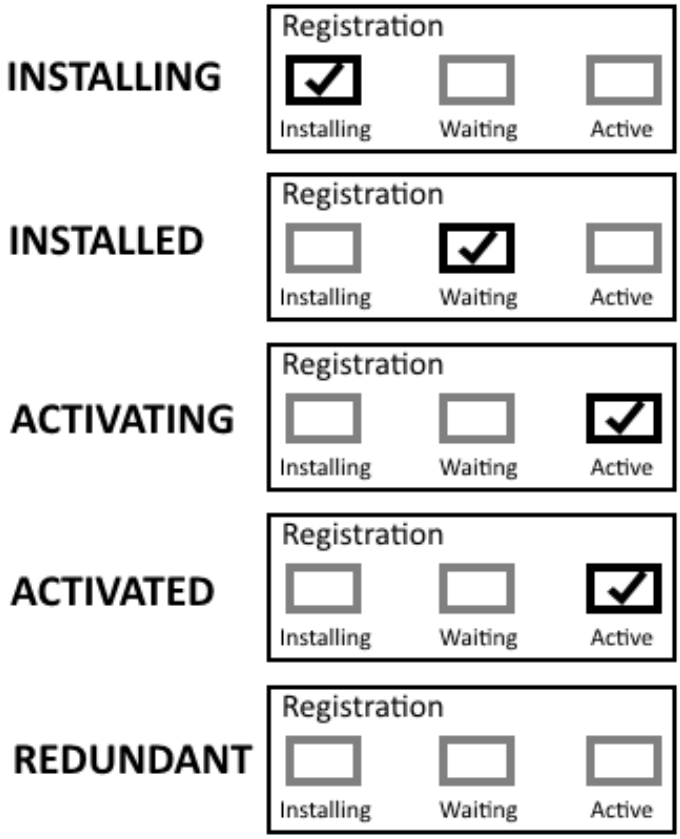


Figure 5.3 *The lifecycle of a service worker based on the Worker lifecycle graph in [40]*

When the initialization of Lsyncr begins, it will register a new service worker. The implementation of a service worker is inside a single JavaScript file. In order to register the service worker the name of the JavaScript file containing the service worker logic must be given to the registration method. Once the registration is done the service worker is installed and activated. If a service worker already exists, nothing is done unless the new service worker differs from the old one. In that case the new service worker will be installed but it will not be activated until all user applications using the old service worker are closed. After they are closed the new service worker will be removed and the new one will replace it. The service worker lifecycle can be seen in the figure 5.3.

When the service worker has been activated, the client can subscribe its endpoint to the push service. The subscription requires the permission of the user. Google Chrome will launch a pop up window containing the question whether the user would like to give permission for the push notifications. When user accepts the notifications, the subscription can be finalized and Google Chrome will then take

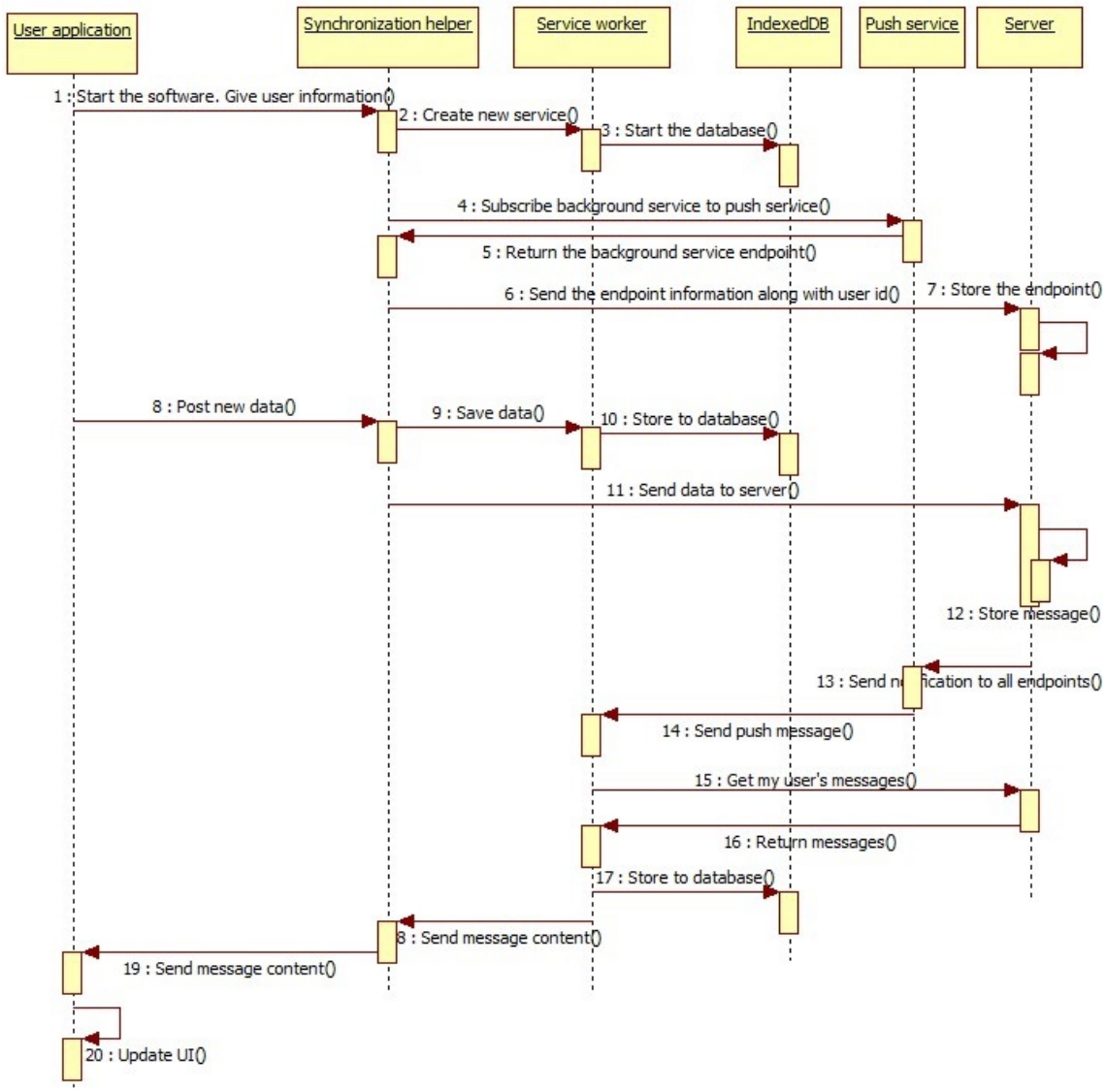


Figure 5.4 *Sequence diagram about the starting of the application and about modifying data by the client.*

```
$("#newMessageInput").keypress(function(e) {
  if (e.keyCode == 13) {
    Lsyncr.getActiveUser().then(function(userId) {
      if (typeof(userId) !== 'undefined' && userId != null) {
        var text = $(e.target).val();
        var id = getMsgId();
        Lsyncr.post(urlList = ["/messages"],
          dataList = [{"id":id, "name": userId, "message":text}],
          true);
      }
    });
  }
});
```

Figure 5.5 *Example of how the chat application creates a new chat message and posts it to Lsyncr.*

care of registering the service worker subscription to the Google App Engine. The client contains a manifest file which has a key that is required for the subscription.

Once the service worker registration is completed Lsyncr will create a Message Channel object and give the port reference of the Message Channel to the service worker. With this reference the service worker can send messages through the Message Channel to Lsyncr. Next Lsyncr gives the service worker the name of the IndexedDB that should be used when accessing the database. The service worker then starts up the IndexedDB if it does not already exist with that name. The service worker registration and subscription was done according to the "Push Notifications on the Open Web" tutorial [11].

When initialization is done, the service worker will start listening to push messages from the push service. Once it receives them it will store the data in the messages to the IndexedDB and forward the data to the Lsyncr which in turn forwards it to the user application. If the user application is closed, Lsyncr is closed as well and it can not receive messages from the service worker. Only the service worker and the IndexedDB instance remain. The service worker can continue to receive messages and modify the contents of the IndexedDB as long as the browser keeps running. A sequence diagram about starting the application and sending new data to the server can be seen in the figure 5.4. Application startup ends at point 7 and data modifying beings at point 8.

```
Lsyncr.post = function(urlList, dataList, setOfflineData) {
  if (setOfflineData) {
    var operations = [];
    for (var i = 0; i < urlList.length; ++i) {
      var url = urlList[i];
      var data = dataList[i];
      var keys = parseUrlForIdb(url);
      operations.push({"keys": keys, "data": data});
    }
    messageChannel.port1.postMessage({ "type":"setData",
    "operations":operations});
  }

  for (var i = 0; i < urlList.length; ++i) {
    var url = urlList[i];
    var data = dataList[i];
    var postHttp = new XMLHttpRequest();
    postHttp.onreadystatechange = function() {
      if (postHttp.readyState==4 && postHttp.status==200) {
        postCallback(postHttp.responseText);
      }
      else if (postHttp.readyState == 4 && postHttp.status != 200) {
        for (var k = 0; k < urlList.length; ++k) {
          messageChannel.port1.postMessage({"type":"saveOfflineMessage",
          "url":urlList[k], "data":dataList[k], "httpType":"post"});
        }
      }
    }
    postHttp.open("post", mainUrl + url, true);
    postHttp.setRequestHeader("Content-type", "application/json");
    postHttp.send(JSON.stringify(data));
  }
}
```

Figure 5.6 *The* ***post*** *method of Lsyncr. User application calls this method to send data to the server.*

Once the user application wants to get data from the server or post new data it will call the correct method of the Lsyncr interface. In the case of a post request the user application calls the **post** method of Lsyncr. A code example of how the chat application calls the **post** method of Lsyncr can be seen in figure 5.5 and how the **post** of Lsyncr is implemented can be seen in figure 5.6. The **parseUrlForIdb** method takes the URL and returns a reversed list of keys. For example if the URL is **/people/2/name** then the parsing method returns a list **[name, 2, people]**. Next Lsyncr sends the processed URL and data to the service worker which stores

```
Lsyncr.get = function(url, getOfflineData) {
  if (getOfflineData) {
    var keys = parseUrlForIdb(url);
    messageChannel.port1.postMessage({ "type":"getData", "keys":keys});
  }
  else {
    var getHttp = new XMLHttpRequest();
    getHttp.onreadystatechange = function() {
      if (getHttp.readyState==4 && getHttp.status==200) {
        getCallback(getHttp.responseText);
      }
      else if (getHttp.readyState == 4 && getHttp.status != 200) {
        getCallback("error");
      }
    }
    getHttp.open("GET", mainUrl + url, true);
    getHttp.send();
  }
}
```

Figure 5.7 *The* ***get*** *method of Lsyncr. User application calls this method to get data from server or database.*

the data to the IndexedDB. Then Lsyncr creates a XmlHttpRequest object and uses it to send a post request to the server. If the request returns back successfully, the post callback method that the user application has registered to will be called. If the request fails, then it is sent to the service worker which stores it into a special offline message array. Every once in a while the offline message array is inspected. If the array contains unsent requests, they are sent to the server. Each time the sending fails the request returns to the array. The request will be removed from the array once the server successfully handles it.

In figure 5.7 is the **get** method of Lsyncr. User application calls this method to get data from either the server or the database. For example in the user application with people and houses the people have friend objects. If the user application wanted to know the name of a specific friend of a specific person then it would call Lsyncr like this:

```
Lsyncr.get("/people/1/friends/3/name", true);
```

This would call the **get** method of Lsyncr which in turn would parse a list of keys from the URL and then call the service worker to fetch the data from the database. I will evaluate this architecture closer in the chapter 6.

```
//Handling messages from synchronization module
var handlePortMessage = function(event) {
  var data = event.data;

  if (data.type == "setData") {
    var transaction = database.transaction([targetObjectStoreName], "readwrite");
    initTransaction(transaction);
    var objectStore = transaction.objectStore(targetObjectStoreName);
    for (var i = 0; i < data.operations.length; ++i) {
      var keyList = data.operations[i].keys;
      var postData = data.operations[i].data;
      setDataToDatabase(keyList, objectStore, postData);
    }
  }

  else if (data.type == "getData") {
    var keyList = data.keys;
    var transaction = database.transaction([targetObjectStoreName], "readonly");
    initTransaction(transaction);
    var objectStore = transaction.objectStore(targetObjectStoreName);
    getFromDatabase(keyList, objectStore);
  }
```

Figure 5.8 *Example of message handling by the service worker.*

In figure 5.8 is an example of how the service worker handles the messages sent by Lsyncr through the Message Channel. All messages have a **type** property and the service worker acts based on the type. In the "setData" and "getData" examples a transaction is created and after that the proper method is called to set data to database or get data from it. These methods will be discussed in the next section.

5.4 IndexedDB

Technical information about the IndexedDB has been discussed in the subsection 3.3.2. This section discusses the IndexedDB implementation regarding the proof of concept application.

The IndexedDB instance is started by the service worker by calling the open method of the IndexedDB interface. The database name and version number are given as parameters to the opening method. If an IndexedDB instance with the same name already exists, the version numbers are compared with each other; if the version numbers are the same then nothing happens and if it is higher in the new one then the IndexedDB will be upgraded to a new version by starting an **onupgradeneeded** event. In the handler of the upgrade event more object stores and indexes can be

```
var configureIndexedDb = function(name) {
  var request = indexedDB.open(name, 3);
  request.onerror = function(event) {
    // Error handling
  };
  request.onsuccess = function(event) {
    database = event.target.result;
    database.onerror = function(event) {
      // Add error handling for database here
    };
    var notifyDatabaseIsReady = { "type":"initDb" };
    messagePort.postMessage(notifyDatabaseIsReady);
  };
  request.onupgradeneeded = function(event) {
    database = event.target.result;
    for (var i = 0; i < activeObjectStores.length; ++i) {
      var containsKey = false;
      for (var k = 0; k < database.objectStoreNames.length; ++k) {
        if (database.objectStoreNames[k] == activeObjectStores[i]) {
          containsKey = true;
          break;
        }
      }
      if (!containsKey) {
        database.createObjectStore(activeObjectStores[i]);
      }
      var objectStore = event.currentTarget.transaction.objectStore(activeObjectStores[i]);
      try {
        objectStore.createIndex("typeIndex", "type", {"unique":false});
      }
      catch (e) {
        // Index with same name already exists in the object store
      }
    }
  };
}
```

Figure 5.9 *The initialization of IndexedDB in service worker.*

added to the IndexedDB and it is important to notice that adding them anywhere else is not possible. Therefore the object stores and indexes should be considered as variables which can not be easily modified so the database should be well designed in order to avoid constant database restarts.

Example code of setting up the IndexedDB can be seen in figure 5.9. The code is executed in the service worker and the **configureIndexedDb** method is called when the service worker receives a command from Lsyncr to initialize the database. The **activeObjectStores** list has names of object stores that are currently in use in the IndexedDB instance. In this example it has only one name. The index is created at the end of the method on the **onupgradeneeded** event handler. Lsyncr is notified through the Message Channel when the database is ready.

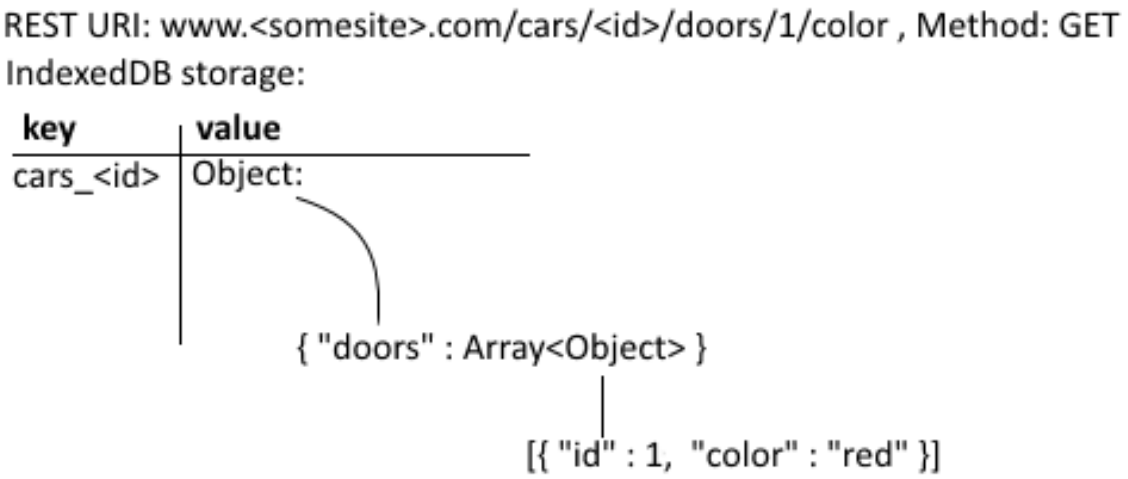


Figure 5.10 *The IndexedDB structure of the application demonstrated. The return value of this call would be "red".*

In the proof of concept application the structure of the database was built so that it would match the resource structure of the REST style. This was done because the data synchronization library was designed to be used by multiple applications without having to use an external schema file. The closer evaluation of this decision is handled in chapter 6.

IndexedDB data records are stored as key-value pairs. In the database structure of this solution the key of the record consists of the first resource key of the REST URL and the id field of the object. This makes it possible to have unique record keys. The value of the record contains the rest of the data as a JavaScript object. It is important to understand that JavaScript objects are like dictionaries so they too have their own key-value pairs. In case the REST URL would be nested then the remaining resource names would be stored as object references to the value of the IndexedDB record with the same name that the resource has in the URL. When the nested URL's final resource name is reached then the value of that resource is set as the value of the final object. This would work both for getting and posting data. An image describing this structure is in figure 5.10. Each resource in the REST URL matches to a variable with a same name in each object with the exception of id-values which organize the objects in case of object arrays.

In figure 5.11 is the method that creates a new data record to IndexedDB or updates an existing one. This method is called from the "handlePortMessage" method which can be seen in figure 5.8 and which was called by Lsyncr through the Message

```
var setDataToDatabase = function(keys, objectStore, data) {
  if (keys.length > 0) {
    var key = keys.pop();
    var id = data.id;
    var firstKey = "";
    if (typeof(id) !== 'undefined' && keys.length == 0) {
      firstKey = key + id;
      data["type"] = key;
    }
    else {
      id = key;
      firstKey = id + keys.pop();
    }
    // See if record with the key exists in ObjectStore
    var request = objectStore.get(firstKey);
    request.onsuccess = function(event) {
      if (typeof(request.result) !== "undefined") {
        // Update existing record
        if (keys.length == 0) {
          var requestUpdate = objectStore.put(data, firstKey);
          initObjectStoreSetRequest(requestUpdate, id);
        }
        else {
          setDataToLastObject(id, firstKey, keys, objectStore, data, request.result);
        }
      }
      else {
        // Create new record
        var requestUpdate = objectStore.add(data, firstKey);
        initObjectStoreSetRequest(requestUpdate, id);
      }
    }
    request.onerror = function(event) {
      // Handle error
    }
  }
```

Figure 5.11 *Method that creates a new record to IndexedDB, updates an existing one or calls a method to update a single attribute of an existing record.*

Channel. The record is first fetched from the database by using the main key and if it is not found then a new record with the new key and data is created. If it is found, based on the length of the REST URL the record is either updated or the "setDataToLastObject" method is called. In this method the key-value tree of the record is traversed and updated based on the key list. After that the record is updated to the IndexedDB after which a notification is sent to Lsyncr through the Message Channel. Lsyncr notifies the user application by calling the method the user application has registered.

All the objects that are related to each other should remain in the same object

```
var setDataToLastObject = function(objectId, firstKey, keys, objectStore, data, currentObject) {
  var helperObject = undefined;
  if (keys.length > 0) {
    helperObject = currentObject;
    while (keys.length > 1) {
      var currentKey = keys.pop();
      if (Array.isArray(helperObject)) {
        for (var i = 0; i < helperObject.length; ++i) {
          if (currentKey == helperObject[i].id) {
            var nextObject = helperObject[i];
            helperObject = nextObject;
            break;
          }
        }
      }
      else {
        var nextObject = helperObject[currentKey];
        helperObject = nextObject;
      }
    }
    helperObject[keys[0]] = data[keys[0]];
  }

  var requestUpdate = objectStore.put(currentObject, firstKey);
  initObjectStoreSetRequest(requestUpdate, objectId);
}
```

Figure 5.12 *Method that updates an existing IndexedDB record. Usually this method is used to update a single attribute of an existing record.*

store because otherwise an IndexedDB transaction could not be kept atomic. A transaction is always bound to a certain object store. In this thesis there is only one object store because dividing different data types to their own object stores inside one application would require a customizable schema which was left out in the implementation of this thesis. In the second user application with "person" and "house" objects the person and house are related to each other. In case a person decided to move to a house both the person and the house record would need to be modified in one transaction. In IndexedDB this can be done by starting a read-write transaction and while the transaction continues to receive operations it will stay alive and block the other transactions from modifying the database.

In order to make fetching data faster the index properties of the IndexedDB were used. Since they need to be created in the **onupgradeneeded** event handler they could not be dynamic. Therefore each record has a **type** field which matches the first resource of the REST URL which can also be found within the key of each record. This way fetching the data of a certain type could be done by finding it with the help of the index belonging to that type.

The way service worker fetches data from IndexedDB can be seen in figure 5.13.

```
var getFromDatabase = function(keys, objectStore) {
  if (keys.length > 0) {
    var rangeKey = keys.pop();
    var typeIndex = objectStore.index("typeIndex");
    var returnData = {"key": rangeKey, "data":[]};
    var range = IDBKeyRange.lowerBound(rangeKey);
    var shouldCursorStop = false;
    typeIndex.openCursor(range).onsuccess = function(event) {
      var cursor = event.target.result;
      if (cursor && cursor.key == rangeKey && !shouldCursorStop) {
        if (keys.length == 0) {
          returnData.data.push(cursor.value);
        }
        else if (keys[keys.length - 1] == cursor.value.id) {
          keys.pop();
          var helperObject = cursor.value;
          while (keys.length > 0) {
            var currentKey = keys.pop();
            if (Array.isArray(helperObject)) {
              for (var i = 0; i < helperObject.length; ++i) {
                if (currentKey == helperObject[i].id) {
                  var nextObject = helperObject[i];
                  helperObject = nextObject;
                  break;
                }
              }
            }
            else {
              var nextObject = helperObject[currentKey];
              helperObject = nextObject;
            }
          }
          returnData.data.push(helperObject);
          shouldCursorStop = true;
        }
        cursor.continue();
      }
      else {
        messagePort.postMessage({"data": returnData, "type":"get"});
      }
    };
  }
}
```

Figure 5.13 *The method that service worker uses to fetch data from IndexedDB.*

```
self.addEventListener('push', function(event) {
  event.waitUntil(
    self.registration.pushManager.getSubscription().then(function(subscription) {
      fetch(syncUrl, {
        method: "post",
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(subscription)
      }).then(function(response) { return response.json(); })
      .then(function(data) {
        handlePortMessage( {"data": {"type":"setData",
            "operations": [{"data": data.data,"keys":data.keys}]}});
        if (messagePort != null && typeof(messagePort) !== 'undefined') {
          messagePort.postMessage({"data": data.data, "type":"update"});
        }
      });
  }));
});
```

Figure 5.14 *The push message listener of service worker. This method is called when the push service sends data to the service worker.*

The index is used to filter the data based on the **type** of the data. Based on the key list either multiple records, a single record or the value of a property of a record is fetched. The **continue** method of cursor moves to the next record on the record list of the cursor. Each time the **continue** is called the execution goes to the beginning of the **onsuccess** method. If the cursor moves too far, it will either become null or it will move to the other data types of the index. In the proof of concept application there are messages, people and houses in the database. If the user wanted to get all the messages then at some point the cursor would start showing records of people and houses. At this point the execution is forced to stop and the fetched data is sent to Lsyncr through the Message Channel.

5.5 Server and push service

The server is made with Node.js and it uses the Express framework [42] to handle the requests from the client. The server stores the user names and endpoints into a dictionary so a single user name may have multiple endpoints. One endpoint represents one device. When a client sends the server new data the server goes through all of its endpoints and sends the push service a new message with the data and the endpoint. Each time the push service receives a message it checks the endpoint and based on the endpoint it forwards the message to the correct client.

The server has all of the data in its memory so when the server reboots all the data is lost. A server side database was not necessary in the scope of this thesis. The server implements a REST interface and it has methods for sending and getting chat messages and for adding and modifying people and houses as well as getting people and house information.

The push service is a simple Google App Engine project that has only the basic functionalities that Google provides by default. Google Chrome requires that the push service of the Push API is a Google App Engine. In order to register the service worker to the push service the client must have an extra manifest file. The file contains the key of the Google App Engine project.

When the server sends data to the push service it goes to `https://gcm-http.googleapis.com/gcm/send` with the endpoint information added to the end of the URI. The authorization key of the Google App Engine project must be sent with each message that is sent to the push service so the push service knows that the sender of the data can be trusted. The authorization key is not supposed to be public information. All the data is sent with the content type set to **application/json**.

In figure 5.14 can be seen the push message handler of the service worker. This method is called each time the push service sends data to the service worker. At the beginning of the push method the service worker sends a synchronization message to the server along with the endpoint of the service worker. In this method the new data is fetched from the server. At the moment the push message can not have any payload because of the way Google Chrome implements the Push API. When the service worker receives the new data from the push service the service worker stores the data to the database and notifies Lsyncr about new data through the Message Channel. In the Message Channel handler of Lsyncr the update method of the user application is called along with the data.

5.6 Abandoned solutions

5.6.1 Data synchronization

At first I thought I would use a BaaS system with the application. However, that would have resulted in a vendor lock-in and since I would have received several features without having to implement or study them myself it was better not to

use a BaaS. I tried Firebase [10] which provides the developers a JavaScript library that helps to use the Firebase server with the client. The offline features and data synchronization worked quite well with Firebase. However, if user first went offline, then modified data and then closed the browser the data would not be synchronized since it was not saved to any storage or kept by a service worker. It was kept only in the memory of the Firebase JavaScript instance.

Another database system which has support for easy data synchronization is PouchDB [43] that is based on CouchDB [4] database system. PouchDB provides the developer a JavaScript library that helps them to interact with a database that understands the CouchDB replication protocol. This was left out because then I could not have used any other database and one important point of the proof of concept application was to keep it as modifiable as possible.

5.6.2 Push notification

PushBullet [44] is software which enables the use of a pushing service but in order to get it working in a browser the user would be required to download it as an extension. In this thesis one point was to get persistent data synchronization working without 3rd party extensions so it was left out.

StrongLoop [50] provides a LoopBack [49] framework which is a set of Node.js modules that make it easier to create a server that supports push notifications. However at least based on the documentation the push notification tutorials were only for Android and iOS. It did not seem important to try StrongLoop with a service worker since the service worker already has a push notification system. In addition StrongLoop is not completely free and the free version probably would not be enough for a real software product. Since the framework is only for Node.js the user cannot freely choose which server they wish to use.

6. EVALUATION

In this chapter the developed data synchronization module Lsyncr will be evaluated from the point of view of how well it handles data synchronization and how it performs in offline state. The different technologies that were used to create the synchronization library will be evaluated as well.

6.1 Lsyncr

Traditionally, a user application would communicate with a server through a XmlHttpRequest object which can be used to send GET, POST, PUT and DELETE operations. The interface between the user application and Lsyncr is meant to mimic this behaviour. The user application is not supposed to communicate directly with the server when using Lsyncr. Because the interface of Lsyncr resembles the traditional communication methods the use of Lsyncr becomes intuitive. The interface of Lsyncr is meant to be REST-like because of the way the IndexedDB is handled. It will be discussed closer in the next section.

Lsyncr is meant to work with a service worker and all the database operations are meant to be handled in the service worker. If an application would already use some local database system, it would have to be migrated to function with the service worker. The use of IndexedDB is not forced by Lsyncr but the service worker that was implemented uses IndexedDB as the local database. The implemented service worker has a fixed interface with Lsyncr and the service worker has some initialization methods that Lsyncr calls so if the same service worker was not used then those methods would need to be implemented. Lsyncr also expects to communicate with the service worker through the Message Channel technology.

The data synchronization works well but it does not have a conflict detection system. All data is simply overwritten to the database. The detection and handling of conflicts was not completely in the scope of this thesis but it would have made

Lsyncr more useful. The lack of conflict handling means that Lsyncr can not be used in a real software project in its current state. It simply works as a proof of concept of how to make liquid software on the web.

Lsyncr acts mostly as a message proxy or dispatcher between the user application, the server and the service worker. When Lsyncr receives a command to send a request to the server, it will do so without modifying the request. The only thing Lsyncr does is add the **main URL** information to the request. The main URL is the address of the server that has been configured to Lsyncr. Before sending the request of the user application to the service worker Lsyncr parses the URL of the request to match the IndexedDB structure that was designed for the proof of concept application. In hindsight this should have been done in the service worker because Lsyncr should not have to care about the database structure. Now an unnecessary dependency exists between Lsyncr and the IndexedDB.

6.2 IndexedDB and Service Workers

The interface of IndexedDB was easy to use and since it can store large amounts of data it is a very good storage system for the browser. The order in which IndexedDB operations are executed might be hard to understand at first. For example when a request to add data to an object store is created the **add** method of the object store instance is called. The **add** method returns a reference to the add request. Now the user can add **oncomplete** and **onerror** on to the request even though based on the order of the commands in the code it would seem that the request has already been executed and adding event handlers to it now would be useless. However, this system works and the event handlers are called when the "add" request actually finishes. This is because of the asynchronous nature of the IndexedDB. The adding of event handlers also clutters the code. The developer should be prepared to build customizable initialization methods for transactions and requests to help keep the code clean and readable.

Another thing a developer should be aware of when using IndexedDB is the way it handles object references. If two objects with a reference to each other are stored to IndexedDB and if one of them is modified in the database then the modification will not be reflected in the other object. In case of circular references the relationship between the objects should be done with unique id values instead of object references. The user has the responsibility to update the data to everywhere it is referenced.

Regarding the proof of concept application I was not happy with how the database structure turned out. To maximize the use of indexes and to be more efficient when using the database it would have been better to build some kind of configuration system instead of the current structure which mimics the resource structure of REST URL's. Now there is only one index which focuses on the type field of the data and one object store which holds all the data. If the configuration was used then multiple indexes and object stores could be used.

In addition to the structure of the IndexedDB, the relationship between the service worker and the IndexedDB is not optimal. Now the service worker does all of the database operations by calling the methods of the IndexedDB instance. Therefore the structure of the IndexedDB is dictated by the service worker. To keep the architecture more modular the handling of IndexedDB instances should have been made in another script that would be imported to the service worker.

At the current implementation the service worker does some presumptions of the keys it receives from Lsyncr when database operations should be done. The keys are traversed from the last to the first by using the **pop** method of the Array object. The last key is always expected to be some general name of a resources such as "messages" or "people". The second key is expected to be an **id** property. After that the keys can be whatever they want but if user is getting data with a long key chain and while traversing the object keys it encounters an Array type it expects the next key to be an **id** property. This is done because of the attempt to mimic the REST style database structure. A configurable database schema would be better if the database operations should be verified.

In the proof of concept implementation the service worker has the biggest responsibility for the offline support. If Lsyncr fails to send a message to the server, the message will be set in a list of offline messages in the service worker. Periodically the service worker will look at the offline message list and once it sees that there is content it will send it to the server by using the fetch method of the service worker interface. If the message fails again it will be placed back in the offline list. This solution disrupts the current architecture. Lsyncr is meant to be the module that posts new data to the server. However, if the service worker would be required to use Lsyncr to send data from the offline message list to the server then the offline messages could not be synchronized once the user closes the web application.

Using the service worker was easy and it functioned very well. The caching feature

is well done and the developer has freedom to add additional logic to it. In the proof of concept implementation, the caching was kept simple and the file names were simply added to a list which was checked each time a resource was fetched from the server to see whether it could be fetched from the cache instead. It was good practice to keep the resources files in the cache and all the persistent data related to the logic of the application in the IndexedDB.

One problem with the service worker is the fact that user can not use a new version of the service worker if they still have a window open that uses the old one. There might be cases where user has multiple windows open in an application so they would be forced to close down all of them if they wanted an update. One way to fix this would be to make multiple service workers and assign them to different pages. Since service workers can import JavaScript files then the service workers do not need to implement everything themselves.

6.3 Push API

With the current implementation of Push API, no data can be sent with a push message. In addition the developer is forced to use Google App Engine as the push service because the push subscription of Google Chrome will not work unless the subscription is done to a Google App Engine project.

The Push API implementation on the proof of concept application worked well even though the data had to be fetched from the server after the push notification was received instead of having the data on the notification itself. It works very well with the service worker and enables data synchronization on the background. The API was also easy to use as it was enough to implement only one method for the push messages.

6.4 Improvement ideas and missing important features

In order to make the data synchronization better and usable for real software projects it would require several new features. The most important feature is data conflict handling and timestamp checking. If several users were to modify data, there should be support for sending the client software a notification of a conflict and add support methods in order to help with handling the conflict.

Intelligent caching would reduce the amount of resources that need to be loaded from the server. Some people only use specific parts of a program so caching the resources of other sites would fill up the memory of the device faster. This is particularly important with mobile devices which have less storage and which sometimes have to be precise about how much Internet bandwidth they use.

The IndexedDB structure would be better with a customizable schema. When the developer knows what data they are storing, they can build better indexes to speed up data queries. Some type checking could also be implemented into the data storage operations because it would make the program less susceptible for errors and bugs in case the server sends wrong kind of data to the user. All around the proof of concept implementation lacks error handling.

7. OUTLOOK OF THE FUTURE

7.1 Push API

I could not find any documents that would have estimated when the Push API is ready. Both the API and the protocol standard are still under development. It is good that Push API requires a service worker to function because that will hopefully direct the future developers to build their software products with distinct modules for server communications, database access and other application functionalities. The use of a push service will also force the developers to use a specific architecture in their software and the idea of having all the active connections of the client to be tied to a push service makes sense because it reduces the load of the main server.

I could not test the Push API data sending features properly because it is not ready. For the Push API to become a good solution for data sending it should support different data types. Server-sent events technology supports only UTF-8 encoded content so Push API should improve from this. The biggest problem at the moment is how fast the standard can be finalized and how fast the project teams behind different browsers will implement the support for Push API and service workers.

7.2 Service Workers

Service worker is already functioning very well at least on the Google Chrome. At the moment it is the only way to have true background synchronization even when the web application is not active in the browser. The fact that it uses its own thread makes it really useful to synchronize data as it does not interfere with the user application.

The service worker will replace AppCache as the offline cache for the browser since AppCache is no longer maintained and the service worker is currently being worked on. The resource cache of the service worker already works very well.

7.3 IndexedDB

It is good that the development of Web SQL has stopped because now there are no competing standards and the developers can focus on learning one tool. It also means that IndexedDB is at the moment the best tool for developers to stores large amounts of data. The wide support for data types and the use of transactions and indexes are what makes IndexedDB such a good database for the client. At the moment IndexedDB interface is not as simple as LocalStorage but implementing a reusable layer on top of it is not too difficult. One JavaScript module that already does this is called LocalForage [32].

When IndexedDB becomes more widely used the amount of local data may increase in web applications. This reduces the amount of data that needs to be downloaded from the server but increases the size of the local storage amounts. For mobile devices this is both good and bad. The bandwidth usage is reduced but the storage size may be quite small for some use cases. It is important how the user can control the limit of storage from the browser and how well the applications can react to a smaller storage size.

7.4 Browser and device support

The biggest problem for Push API, service workers and IndexedDB is the lack of support in several browsers. Microsoft has often been very slow in implementing anything new to their browsers so they might again hinder the use of these new tools because many people use IE or Edge. If a developer wants to have full browser support for their web applications, they will not be able to use the new tools and instead have to come up with different means of implementing their features or simply drop the support of specific features for specific browsers.

The concept of Internet of Things is gaining popularity. The browser or web runtime would have to be able to run in very different devices if all the applications in an IoT environment were web applications. Even though the browser can be used to create liquid software it should not be thought of as a complete replacement of native applications. Some devices such as smart fridges or smart televisions might need the native application to work properly in an IoT environment as their clients require an access to low level functions which the browser can not use.

It fits better to have browsers together with the native applications even though it forces the user to develop their application on several platforms. The server would always be the same for all clients so most of the logic could be handled there. If the server handled a large amount of the logic, the applications on the devices would require less development. Of course if the device can run a modern browser then it would be efficient for it to use that as the client software.

8. CONCLUSIONS

Liquid software is a computer program that is designed to work with multiple devices in multiple environments. The main idea of liquid software is that user can use the application with different devices sequentially or simultaneously and there can be more than one user using the same application. The data between the applications is synchronized to all devices and switching from one device to another is completely hassle-free.

This thesis focused on liquid software from the point of view of web applications that run on a web browser. To investigate how such web applications could be made liquid a proof of concept application was made. Different browser technologies were studied and some very new ones were chosen to help in the implementation of the software: service workers, Push API and IndexedDB along with some older technology such as Node.js and basic JavaScript. The main focus of the proof of concept application itself was to study how persistent data can be synchronized between multiple application instances and between the server.

Even though browsers do not have access to system level operations on the computer such as file handling the liquid software was still possible to implement on the browser. The standardization of the technologies that were used is not yet complete and the support for different browsers is still very limited. However, they are the technologies that are seen as the future of web applications and they are constantly being developed by W3C, WHATWG and IETF.

In the future the concept of Internet of things will gains more popularity and more devices start to have computational logic and an Internet access. The devices have different limitations to their processing power, memory, storage size, battery life and display capability and they must all be taken to account when developing liquid software solutions.

BIBLIOGRAPHY

[1] "Phonegap homepage," Adobe Systems Inc., April 2016, Available: http://phonegap.com/.

[2] "Sync across fire kindle devices and apps," Amazon, March 2016, Available: http://www.amazon.com/gp/help/customer/display.html?nodeId=200911660.

[3] "Anypresence homepage," AnyPresence, Inc., April 2016, Available: http://www.anypresence.com/.

[4] "Couchdb homepage," Apache, April 2016, Available: http://couchdb.apache.org/.

[5] "Apache cordova homepage," The Apache Software Foundation, April 2016, Available: https://cordova.apache.org/.

[6] "ios homepage," Apple Inc., April 2016, Available: http://www.apple.com/ios/.

[7] "Osx homepage," Apple Inc., April 2016, Available: http://www.apple.com/osx/.

[8] A. Deveria, "Can i use ... ?" March 2016, Available: http://caniuse.com.

[9] J. Dooley, *Software Development and Professional Practice*. Apress, 2011, Chapter 5.

[10] "Firebase homepage," Firebase, April 2016, Available: https://www.firebase.com/.

[11] M. Gaunt, "Push notifications on the open web," Google Developers, April 2016, Available: https://developers.google.com/web/updates/2015/03/push-notifications-on-the-open-web.

[12] "Welcome to gmail," Google, March 2016, Available: https://www.google.com/intl/en/mail/help/about.html.

[13] "Android homepage," Google Inc., April 2016, Available: https://www.android.com/.

[14] "Google app engine documentation," Google Inc., May 2016, Available: https://cloud.google.com/appengine/docs.

[15] G. Harrison, *Next Generation Databases: NoSQL, NewSQL, and Big Data.* Apress, 2015, Chapter 1.

[16] "The websocket protocol," Internet Engineering Task Force, December 2011, Available: http://tools.ietf.org/html/rfc6455.

[17] "Hypertext transfer protocol version 2 (http/2) proposed standard," Internet Engineering Task Force, May 2015, Available: https://tools.ietf.org/html/rfc7540.

[18] "Generic event delivery using http push internet-draft v4," Internet Engineering Task Force, March 2016, Available: https://tools.ietf.org/html/draft-ietf-webpush-protocol-04.

[19] "Node.js homepage," Joyent, Inc, April 2016, Available: https://nodejs.org/en/.

[20] "jquery homepage," The jQuery Foundation, April 2016, Available: https://jquery.com/.

[21] "Kii homepage," Kii, Inc., April 2016, Available: https://en.kii.com/.

[22] "Kinvey homepage," Kinvey, Inc., April 2016, Available: http://www.kinvey.com/.

[23] E. Kitamura, "Working with quota on mobile browsers," January 2014, Available: http://www.html5rocks.com/en/tutorials/offline/quota-research/.

[24] "Kumulos homepage," Kumulos, April 2016, Available: https://www.kumulos.com/.

[25] "Linux kernel homepage," Linux Kernel Organization, Inc., April 2016, Available: https://www.kernel.org/.

[26] P. Lubbers, "How html5 web sockets interact with proxy servers," March 2010, Available: http://www.infoq.com/articles/Web-Sockets-Proxy-Servers.

[27] D. C. Marinescu, *Cloud Computing: Theory and Practice.* Morgan Kaufmann Publishers, 2013, Chapter 2.

[28] "Introducing xbox smartglass," Microsoft, March 2016, Available: http://www.xbox.com/en-US/smartglass.

[29] "Microsoft edge browser homepage," Microsoft, May 2016, Available: https://www.microsoft.com/en-us/windows/microsoft-edge.

[30] "Windows 10 mobile homepage," Microsoft, April 2016, Available: https://www.microsoft.com/en-us/mobile/windows10/.

[31] T. Mikkonen, K. Systä, and C. Pautasso, "Towards liquid web applications," *ICWE 2015*, 2015.

[32] "Localforage homepage," Mozilla, May 2016, Available: https://mozilla.github.io/localForage/.

[33] "Channel messaging api," Mozilla Developer Network and individual contributors, November 2015, Available: https://developer.mozilla.org/en-US/docs/Web/API/Channel_Messaging_API.

[34] "The structured clone algorithm," Mozilla Developer Network and individual contributors, August 2015, Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm.

[35] "Indexeddb basic concepts," Mozilla Developer Network and individual contributors, January 2016, Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB.

[36] "Service worker api," Mozilla Developer Network and individual contributors, March 2016, Available: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[37] "Serviceworker," Mozilla Developer Network and individual contributors, April 2016, Available: https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker.

[38] "ServiceWorkerContainer," Mozilla Developer Network and individual contributors, April 2016, Available: https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerContainer.

[39] "ServiceWorkerRegistration," Mozilla Developer Network and individual contributors, April 2016, Available: https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration.

[40] "Using service workers," Mozilla Developer Network and individual contributors, March 2016, Available: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers.

[41] "Using the notifications api," Mozilla Developer Network and individual contributors, March 2016, Available: https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API/Using_the_Notifications_API.

[42] "Express homepage," Node.js Foundation, April 2016, Available: http://expressjs.com/.

[43] "Pouchdb homepage," April 2016, Available: https://pouchdb.com/.

[44] "Pushbullet homepage," PushBullet, April 2016, Available: https://www.pushbullet.com/.

[45] "Qt homepage," The Qt Company, April 2016, Available: https://www.qt.io/.

[46] "The navigator object," Refsnes Data, April 2016, Available: http://www.w3schools.com/jsref/obj_navigator.asp.

[47] "Understanding the basics of backend as a service (baas)," SiliconIndia, August 2012, Available: http://mobile.siliconindia.com/news/Understanding-the-Basics-of-Backend-as-a-Service-BaaS-nid-126045.html.

[48] *Storage Concepts: Storing and Managing Digital Data.* Hitachi Data Systems Academy, Hitachi Data Systems, 2012, Chapter 1.

[49] "Loopback documentation," StrongLoop, April 2016, Available: https://docs.strongloop.com/display/public/LB/LoopBack.

[50] "Strongloop homepage," StrongLoop, April 2016, Available: https://strongloop.com/.

[51] A. Taivalsaari, T. Mikkonen, and K. Systä, "Liquid software manifesto: The era of multiple device ownership and its implications for software architecture," vol. 6, 2013.

[52] "Telegram FAQ," Telegram Messenger LLP, March 2016, Available: https://telegram.org/faq.

[53] "How to choose the right backend as a service (baas) platform," Waracle, February 2016, Available: http://waracle.net/how-to-choose-the-right-backend-as-a-service-baas-platform/.

[54] "Html living standard, offline web applications," The Web Hypertext Application Technology Working Group, March 2016, Available: https://html.spec.whatwg.org/multipage/browsers.html#offline.

[55] "Html living standard, server-sent events," The Web Hypertext Application Technology Working Group, March 2016, Available: https://html.spec.whatwg.org/multipage/comms.html#server-sent-events.

[56] "Html living standard, web sockets," The Web Hypertext Application Technology Working Group, March 2016, Available: https://html.spec.whatwg.org/multipage/comms.html#websocket.

[57] "Html living standard, xmlhttprequest," The Web Hypertext Application Technology Working Group, March 2016, Available: https://xhr.spec.whatwg.org/.

[58] "Living standard, notifications api," The Web Hypertext Application Technology Working Group, March 2016, Available: https://notifications.spec.whatwg.org/.

[59] "Windows homepage."

[60] "Worker."

[61] "Same origin policy," The World Wide Web Consortium, January 2010, Available: https://www.w3.org/Security/wiki/Same_Origin_Policy.

[62] "Indexed database api, w3c recommendation," The World Wide Web Consortium, January 2015, Available: https://www.w3.org/TR/IndexedDB/.

[63] "Push api: W3c working draft 15 december 2015," The World Wide Web Consortium, December 2015, Available: https://www.w3.org/TR/2015/WD-push-api-20151215/.

[64] "Service workers, w3c working draft," The World Wide Web Consortium, June 2015, Available: https://www.w3.org/TR/service-workers/.

[65] "Web storage (second edition)," The World Wide Web Consortium, November 2015, Available: https://www.w3.org/TR/webstorage/.

[66] "Web Storage (Second Edition), W3C Recommendation," The World Wide Web Consortium, April 2016, Available: https://www.w3.org/TR/2016/REC-webstorage-20160419/.

[67] "Web workers: Working draft," The World Wide Web Consortium, April 2016, Available: https://www.w3.org/TR/2015/WD-workers-20150924/.

[68] "The world wide web consortium homepage," The World Wide Web Consortium, April 2016, Available: https://www.w3.org/.