Department of Computer Science

University of Kaiserslautern

Master Thesis

Offline caching in web applications for AntidoteDB

Server Khalilov

red17electro@gmail.com

University of Kaiserslautern
Department of Computer Science
Software Engineering

Leader:
Prof. Dr. Arnd Poetzsch-Heffter

Supervisor:
Dr. rer. nat. Annette Bieniusa

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Zusammenfassung

Zusammenfassung auf deutsch

Abstract

The purpose of this thesis is to explore the possibilities of developing an offline web application, which would serve as a client for the AntidoteDB database. We developed a prototype of the application and designed the architecture of the application in such a way that both offline and online functionalities are possible.

To model the data stored offline in the local database of a web-browser, Conflict-free Replicated Data Types (CRDTs). It lets to ease the task of merging the changes.

Apart from that, the client-server protocol was designed, in order to support the functionality of the application. The paper could be divided into two parts: firstly, the problem of designing mentioned solution is going to be discussed. Secondly, there is an implementation part and a description of how specific problems were tackled.

Acknowledgement

I would like to take this opportunity and express my biggest gratitude to my friends and everyone, who supported me through all the ups and downs I had during my time at the Univesity of Kaiserslautern.

Firstly, I am very grateful to Prof. Dr. Arnd Poetzsch-Heffter and my thesis supervisor Dr. rer. nat. Annette Bieniusa for making it possible for me to work on the topic that suits my area of interest. Moreover, I want to thank Dr. rer. nat. Annette Bieniusa for having the door to her office always open for a discussion, whenever I needed it. I could not ask for more.

Next, I would like to thank the stuff at Software Technology Group, who made me feel very welcome and patiently answered any questions I had. Especially, Mathias Weber, Peter Zeller, and Deepthi Akkoorath.

Finally, I want to mention my family and, particularly, my parents. They did their very best to encourage and motivate me throughout the whole period of my study in Germany. Nothing of it would ever be possible without them. Thank you.

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema „Offline caching in web applications for AntidoteDB“ selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle kenntlich gemacht.

Kaiserslautern, den 15. December 2018 Server Khalilov

Contents

1 Introduction

In this chapter we are going to discuss the motivation, research questions and the scope of this thesis.

1.1 Motivation

The main motivation of this thesis is to explore the possibillities of implementing a web-client for the AntidoteDB with a support of caching and by utilizing the main features of the database. As AntidoteDB is already known for its use in the development of applications, it would dramatically improve the user-experience if offline work with the database is going to be reached. We are going to explore the best ways to develop such an application, design it and implement. The goal of this thesis is to show you that it is possible to build an application based on Antidote-DB that works both online and offline. Here and from now on, we will call the application we are going to develop – WebCache.

1.2 Research questions

The following research questions are going to be addressed in this thesis:

- **RQ1.** How efficient is it to use a web-client with cache rather than without it?
- **RQ2.** What are the methods available to implement web-applications that would be able to work off-line and in the conditions of poor network connections?
- **RQ3.** What could be a scalable solution for transmitting CRDT data between a server and clients?

1.3 The structure

Here, I am going to briefly sketch the structure of this thesis and explain the contents of each chapter.

- Description of the main requirements of the application;
- Design of the architecture
- Description of the implementation phase
- Evaluation
- Conclusion

2 Theoretical background

In this chapter, we are going to introduce the reader to the theoretical concepts, which represent a prerequisite to have the understanding of this thesis.

2.1 Main concepts

Distributed database is "a collection of multiple, logically interrelated databases distributed over a computer network"[1]. A *geo-distributed database*, in its turn, is a database, which is spread across two or more geographically distinct locations and runs without experiencing performance delays. The maintaining of such databases brings its challenges. As the database is spread across several locations, there should be a replication process in order to ensure that replicas of that database synchonize and have the latest state of the data. This replication process should be fast, because if there are two replicas of the database, then whenever there is some information written to the first replica, it should be accessable to users, who use the second one. That is the problem of the availability, but before the information at replicas becomes available, it first should be checked over the consistency, as the states of the replicas should be equal. That is a complex task to solve.

Working with such a distributed database, whenever the data is needed to be read or changed in any way, a transaction should be started, executed and closed. A *transaction* is a basic unit of computing, which consists of sequence of operations that are applied atomically on a database. Transactions transform a consistent database state to another consistent database state. A transaction is considered to be *correct*, if it obeys the rules, specified on the database. As long as each transaction is correct, a database guarantees that concurrent execution of user transactions will not violate database consistency [1]. *Consistency* requires transactions to change the data only according to the specified rules. An example of consistency rule can be the following: let us say that in a bank database the bank account number should only consist of integer numbers. If an employee tries to create an account that contains something other than integer numbers in it, then the database consistency rule will disallow it. Consistency rules are important as they control the incoming data and reject the information, which does not fit.

Sequential consistency and *linearizability* are two consistency conditions that are well-known. Sequential consistency requires that all of the data operations appear to have executed atomically (i.e. independently), in some sequential order. When this order must also preserve the global ordering of non-overlapping operations, this consistency is called linearizability[2]. Linearizability guarantees that the same operations are applied in the same order to every replica of the data item[3]. Serializability is a guarantee about transactions, that they are executed serially (i.e. without overlapping in time) on every set of the data items[3]. Serializability is more strict than sequential consistency, as the granularity of sequential consistency is a single operation, while for serializability it is a transaction. As a result, when serializability is satisfied, the sequential consistency is also satisfied, but not vice versa.

Now, let us introduce different consistency models and the one we will follow in the designing part of WebCache.

As stated by Shapiro [4], "*strong consistency* model could be described in the following way: whenever the update is performed, everyone knows about it". It means that there is a total order of updates and reads are guaranteed to return the latest data, irregardless of which replica is the source of the response. The advantage of strong consistency is that the database is always in a consistent state and to disadvantages we can add low latency, as there is a delay for making sure that all the replicas are in a consistent state before any other read / write requests could be processed. The latency point is a huge drawback for the performance, especially if strong consistency model is considered to be used as a solution for the web, where users usually expect high responsiveness and availability.

The main point of replicating data is to improve such aspects as reliability, availability, performance and latency. However, according to CAP Theorem[5], a distributed database can only have two of the three properties: consistency, availability and partition tolerance. This theorem is very important, as it makes people think towards the trade-off between those three properties for a specific use case. There are some weaker consistency models, where the results of requests can alter depending on the replica[6]. In this thesis, we will stick with partial *causal consistency*. As it is stated in Zawirski et al. [7], causal consistency is "the strongest available and convergent model". They continue their statement saying that "under causal consistency, every process observes a monotonically non-decreasing set of updates that includes its own updates, in an order that

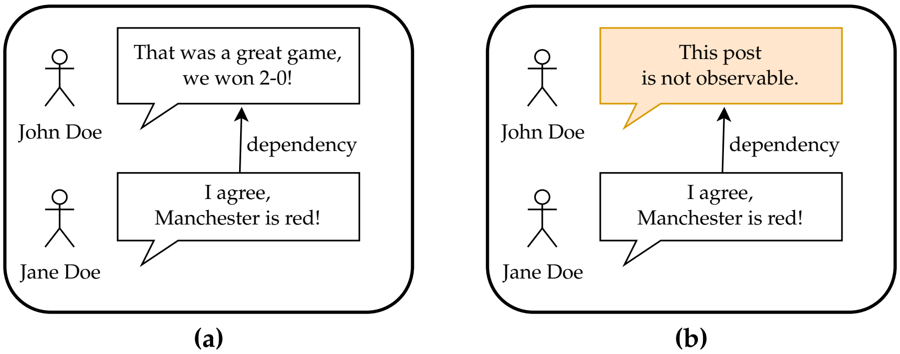


Figure 2.1: An example of how a causally consistent behaviour (a) and the one that is not (b) could work in a social network.

respects the causality between operations". As the causal ordering is respected, it makes it easier for programmers to reason, as it gives the guarantee that related events are visible in the order of occurence, while the events, which have no relation to each other, can be in different order in different replicas. Let us consider an example of an application for some of social netwoks. There, a reply to a wall past happens after the original post is pubslihed. Thus, users should not see the reply before the original post is observable. This type of guarantees are provided by causal consistency. Looking at the **Figure 2.1** you can see that on the left subfigure, the user can see the original wall post as well as the reply, while on the right subfigure, without causal consistency, the user sees only the reply, while the original wall post is missing.

2.2 AntidoteDB

For this thesis, one of the core parts in the architecture of the WebCache belongs to the database called AntidoteDB[8]. It helps programmers to write correct applications and has the same performance and horizontal scalability as AP / NoSQL[9], while it also:

- is geo-distributed, which means that the datacenters of AntidoteDB could be spread across anywhere in the world;
- groups operations into atomic transactions[10, 11];
- delivers updates in a causal order and merges concurrent operations;

Merging concurrent operations is possible because of Conflict-Free Replicated Datatypes (CRDTs) [12], which are used in AntidoteDB. It supports counters, sets, maps, multi-value registers and other types of data that are designed to work correctly in the presence of concurrent updates and failures. The usage of CRDTs allows the programmer to avoid problems that are common for other databases like NoSQL, which are fast and available, but hard to program against[11]. We will cover the topic of CRDTs later in this chapter.

Apart from that, to replicate the data AntidoteDB implements the *Cure*[11] protocol. It is a highly scalable protocol, which provides causal consistency.

To ensure the guarantees it offers, AntidoteDB uses timestamps, indicating the time after the transaction. Timestamps are considered to be unique, totally ordered, and consistent with causal order, which means that if *operation 1* happened before *operation 2*, then the timestamp related to the *operation 2* is greater than the one related to the *operation 1*[12].Whenever the update operation has to be applied, it is also possible to provide a minimum time from what that update should be performed. This information is useful, when a client is working with a server, which is based on AntidoteDB. In such cases, when one data center stops working, the client can reconnect to another one. As a client can remember the latest timestamp for the data it has worked on, the failover to another data center is possible without any additional efforts, as the timestamp information will help the client to request only the data, which has timestamps greater than the one, which is already stored at the client.

2.3 Conflict-Free Replicated Datatypes

As it is stated in the work of Preguiça et al. [13], a conflict-free replicated datatype (CRDT) is an abstract datatype, which is designed for a possibility to be replicated at multiple processes and possesses the following properties:

- The data at any replica can be modified independently of other replicas
- Replicas deterministically converge to the same state when they received the same updates

Replication is a fundamental concept of distributed systems, well studied by the distributed algorithms community[12]. There are two models of replication that are considered: state-based and operation-based. We are going to introduce our reader to both of them below.

Operation-based replication approach

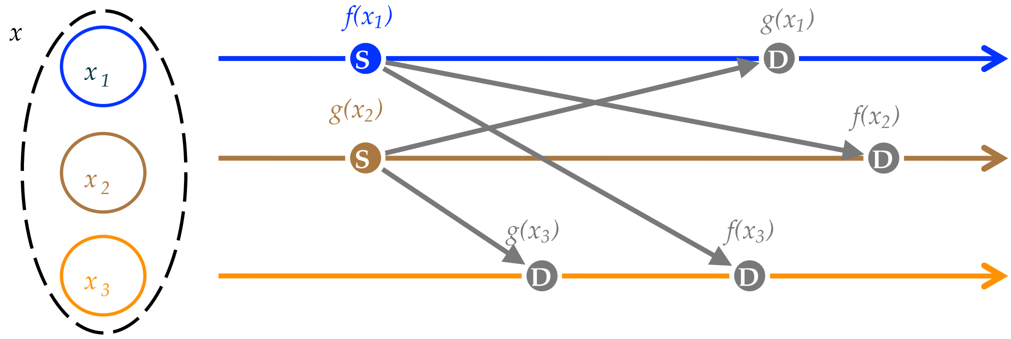


Figure 2.2: Operation-based approach[12]. «S» stands for source replicas and «D» for downstream replicas.

In this thesis, we are going to use the operation-based replication approach, where replicas converge by propagating operations to every other replica[13]. Once an operation is received in a replica, it is applied locally. Afterwards, all replicas would possess all of the updates. At the **Figure 2.2**, you can see that firstly operations $f(x_i)$, $g(x_i)$ applied locally at source replicas x_i, and then the operations are conveyed to all the other replicas. The second part of this process is named *downstream* execution.

This replication approach infers that replicas do not exchange full states with each other, which is a positive in terms of efficiency. Not always, though, as it depends on the task. Sometimes, applying multiple operations at every replica could be costly and this is where state-based replication approach is beneficial.

State-based replication approach

The idea of this approach is kind of opposite to the operation-based one. Here, every replica, when it receives an update, first applies it locally. Afterwards, it sends its updated state to other replicas. Following this way, every replica sends its current full state to other replicas. Afterwards, the merge function is applied between a local state and a received state, and

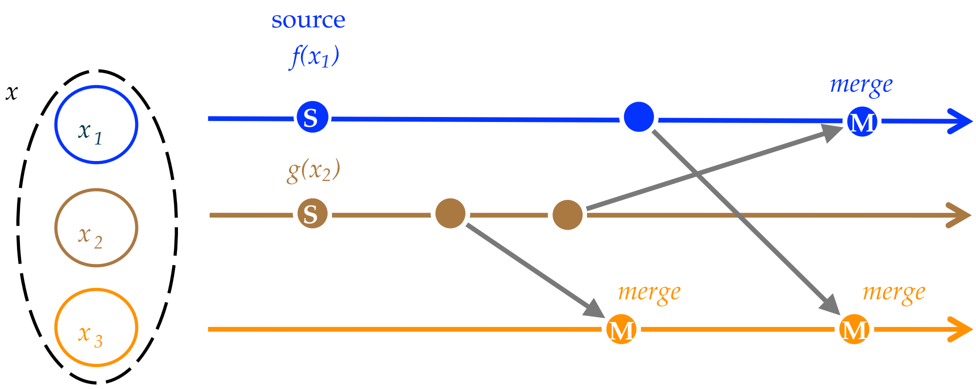


Figure 2.3: State-based approach[12]. «S» stands for source replicas and «M» for merging stages.

every update eventually is going to appear at every other replica in the system. You can see this it at the **Figure** 2.3, where x_i stands for replicas, $f(x_i)$, $g(x_i)$ stands for the functions that apply updates locally at source replicas before sending a new full state for a follow-up merges to other replicas in the system.

3 Requirements

In this chapter, we will describe the requirements for the desired web-application.

3.1 Problem Description

The application should showcase the functionality of communicating with an AntidoteDB database while online. As well as that, it should be reliable, which means it should load instantly and work even in uncertain network conditions. The application should be able to respond quickly to user interactions, be highly available (possible to work with it even when there is no internet at all).

4 Design

As we specified the requirements, we can go further into the design part of the system, and that is what we are going to explain in this chapter. First of all, we will define the protocol of the system. And, secondly, we will describe the client- and server- sides of it.

4.1 Overview of the protocol

The fundamental part of the WebCache will be its protocol design. We need to examine the following cases, in order to specify the protocol of the application:

- A client receives an update from the server
- A client sends an update to the server
- Two clients interact with a server

4.1.1 The way of exchaning the data

Because the AntidoteDB is using CRDT datatypes, the following options are possible to update the database: state-based and operation-based. Due to the time constraints and the amount of work, this thesis will consider only the operation-based approach, which was described in the **Chapter 2**. Therefore, whenever a client wants to update the database, it will send to the server a list of operations. However, whenever it wants to read the value, it will receive the current state of the object from the database.

For this thesis, we are going to use such datatypes as counters and sets. As was already mentioned in the **Chapter 1**, the counter is a number datatype, which allows incrementing and decrementing the value. Reading a counter will return the aggregated value of all performed operations.

<INCLUDE brief description of sets and reference to the chapter one>

4.1.2 A client receives an update from the server

Let us say that a client requests an update from the server. In this case, if the request is successful, the server is going to respond with a value for the requested key and the timestamp of the last write – t_0.

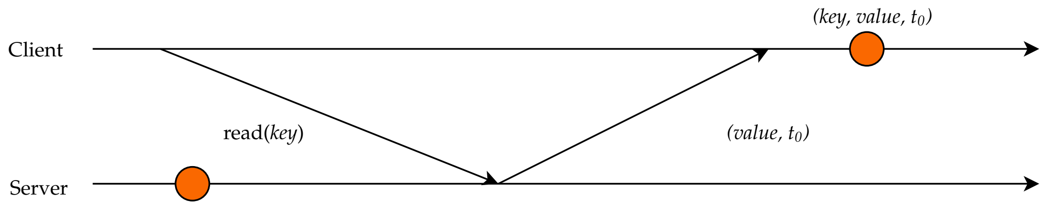


Figure 4.1: The way a client receives a key-value update from the server.

In case a client receives an update from the server, and that update has an earlier timestamp than the one, which is already stored in the client's cache, then a client skips it and can tries later for fresher updates. The implementation of this will be explained later in the chapter of implementation.

4.1.3 A client sends an update to the server

In the case of writing the information to the server, a client has to send a key with an operation to the server. Then the acknowledgment with a timestamp is going to be sent back to the client.

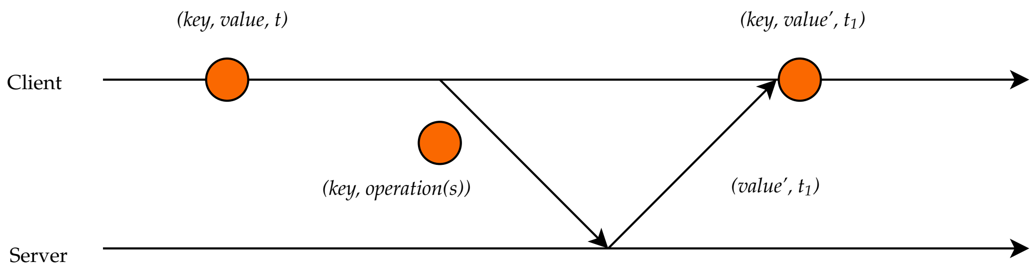


Figure 4.2: The way a client sends a write operation to the server.

Several updates on a client while offline

The client is capable of performing updates when offline. These updates will take effect immediately. However, in order for them to be applied to the server, the client has to be back online. Once the connection is established again, all the updates that were performed on the client-side in offline mode will be sent to the server.

4.1.4 Two clients interact with a server.

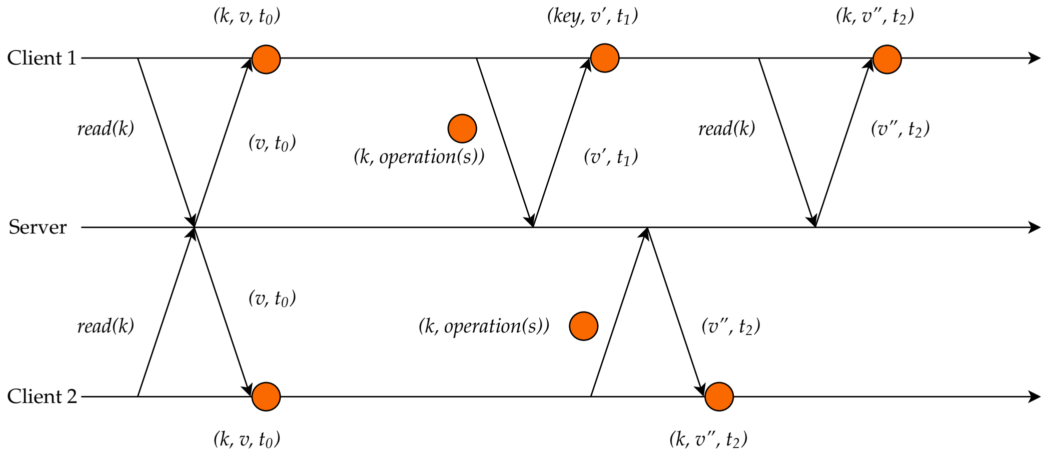


Figure 4.3: Graphical representation over a possible communicaton between two clients and a server.

Let us assume that initially, the server has the key-value pair *(k, v)* at the timestamp t_0. Therefore, after both clients receive updates from the server, they consist of the *(k, v)* pair at the timestamp t_0. In case clients change the value under mentioned key to something else, they will have to get an acknowledgement from the server in order to receive a unique timestamp related to the change. At the representation above, a *Client 1* is acting first and getting an acknowledgement of its change at the time t_1, while Client *Client 2* makes the change later at time t_2. That makes *Client 1* receive a new value v'', when it reads the information from the server again.

4.2 System's main components

As the protocol is explained, we can start designing the system. Let us have a look at the main components of it:

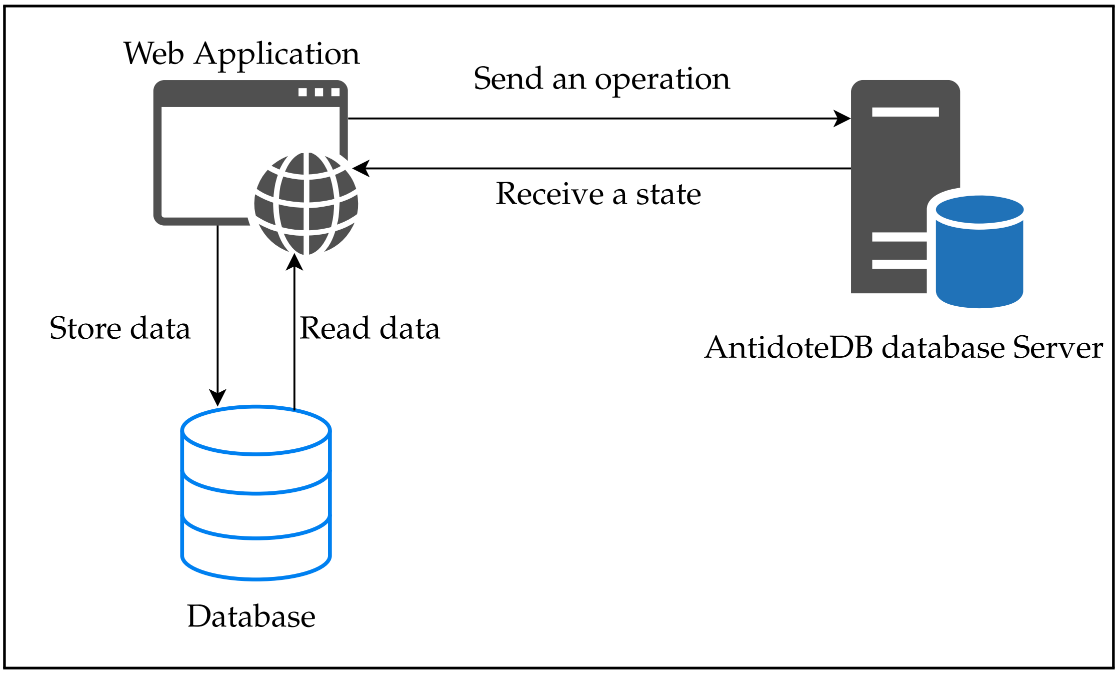


Figure 4.4: A top-level view of the system's design.

4.2.1 Web Application

First of all, in order for the user to be able to communicate with an AntidoteDB server, we are going to have a running web application that serves as a client. It runs in the web-browser, supports various commands from the user and sits on top of the local database layer.

These are the supported commands:

- *read(key)* – an asynchronous function that pulls database changes concerning the *key* that the user passed.

- *update(key, op)* – an asynchronous function that processes user-made updates.
  - *key*: a key, which is going to be updated;
  - *op*: an operation, which is going to be performed on the key;

4.2.2 Server

It is a configured AntidoteDB server that supports the following scenarios:

- receiving an operations or an array of operations performed on a CRDT-object, according to the key, and applying them on the server;
- sending back to the client the state of requested CRDT-object / objects according to their state on the server;

4.2.3 A database layer

This layer should consist of the database, which is going to store the actual states of CRDT-objects, as well as user-added operations performed on the states. When a user performs *read* by *key* operation from the cache, the following actions are taking place:

- Firstly, the state of the object ***O*** is going to be found by ***key*** in the database
- Next, operations ***o*** performed on the object ***O*** are selected
- Afterwards, selected operations ***o*** applied on the object ***O***.
- Finally, the object from the previous step is returned back as a response to the application.

4.3 Offline functionality

In this section, we are going to describe the offline functionality of the system.

Initially, the database is empty. Therefore, if the user is offline from the very beginning, he should be able to add the data into the database himself. The system represented in **Figure** 4.4 will change by having only the Web Application and the database. However, whenever the connection is established with the server, the operations, which were stored in local database while offline, will be sent to the server.

At the **Figure** 4.5, the sequence of getting the data from the local database is shown. This case describes the scenario, when the server is unavailable and the application has to read the value from the local database.

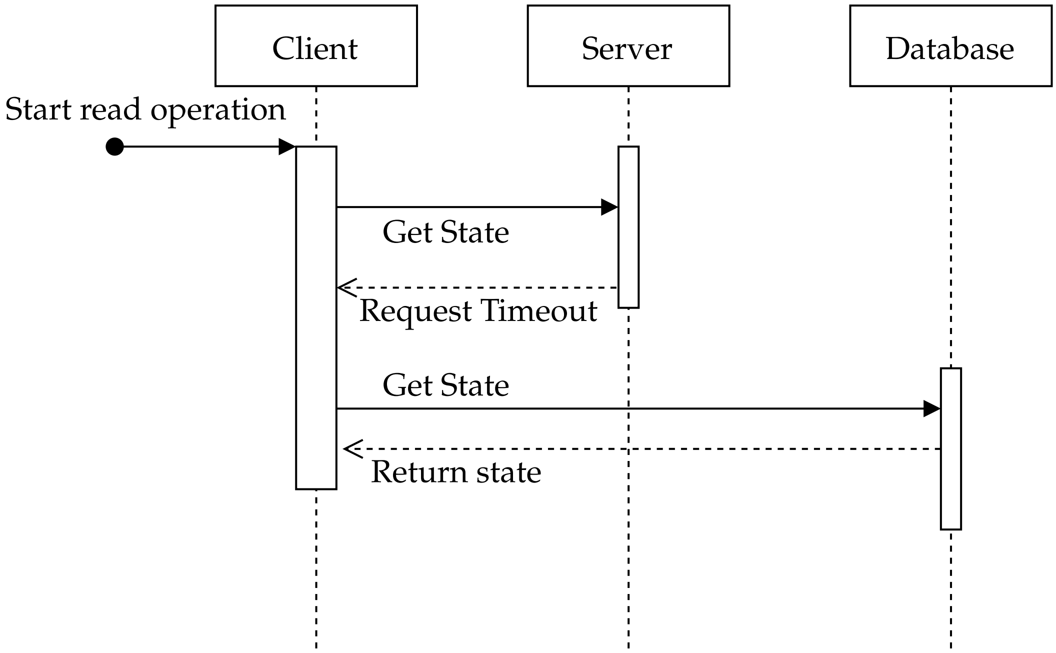


Figure 4.5: Successful request of state from the local database – Sequence diagram.

At the **Figure** 4.6, the sequence of storing the data locally is shown. When the connection is not there, the application will store in the database all offline-performed operations by the user. Afterwards, once the connection is re-established, that data will be sent back to the server. At the point when we read the state again from the server, the data, which is stored locally could be easily removed due to be pointless to hold on it.

4.4 Online functionality

In this section, we are going to describe the online functionality of the system.

4.5 The transition between offline and online modes

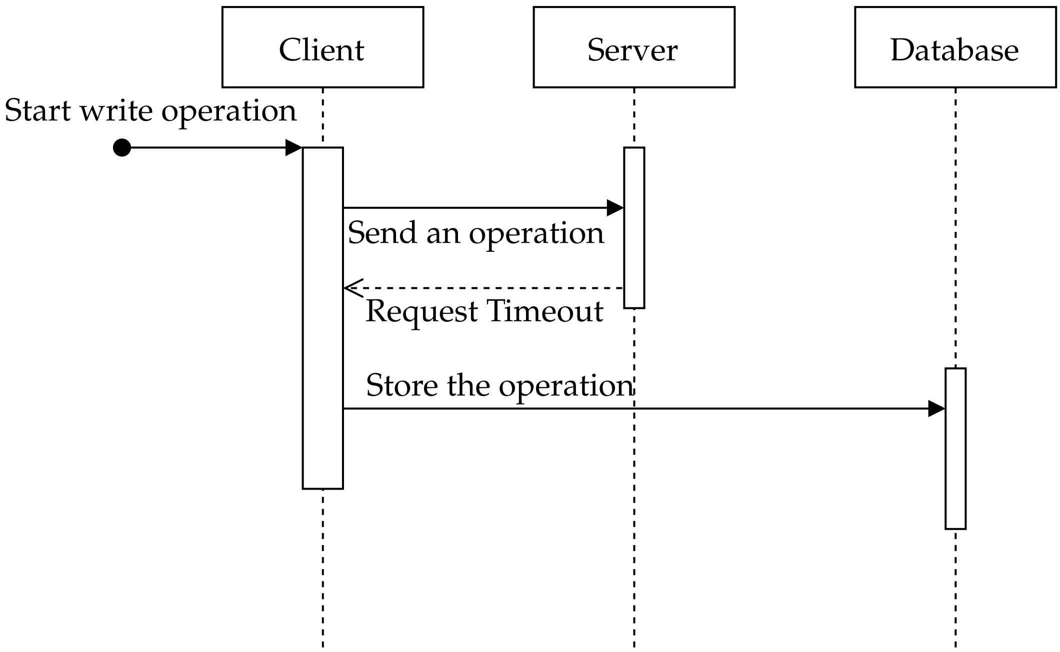


Figure 4.6: Successful storing of an operation in the local database – Sequence diagram.

5 Technologies

This chapter consists of a detailed description of used technologies to implement the design we introduced our reader to in the previous chapter. To give an overview, we believe that the implementation of WebCache's design requires the following components.

- A Cache API for storing assets and make the application available offline.
- A service worker, in order to control requests coming from the client.
- A database to store the data locally.

5.1 Cache API

If we want to store somewhere the HTML, the CSS and the JavaScript, as well as such assets as images, web fonts, then there is a place for all of it – Cache API. It allows conveniently manage the content for the offline use.

5.2 Service Workers

A service worker[14] is a JavaScript file that sits between the application and network requests. As well as that, it runs separately from the page. It has the following properties:

- It is not visible to the user.
- It cannot access the DOM, but it does control pages.
- It can intercept requests as the browser makes them.
  - it can just let requests go to the network as usual.
  - it can skip the network and redirect the request to a cache.
  - or it is possible to perform any combination of these things.

A service worker will be used in order to intercept the network traffic. In situations, when there are problems with network, we are going to get content from cache (which consists of the last content, which we were able to get from the network). However, we will have to wait for the network to fail, before showing the cache-content. And here the problem arises: if the connection is slow, the user would still get frustrating hard-time of waiting for the response. Therefore, we will have to introduce offline-first approach: it means that we will try to get as much information from the cache as possible.

We might still go to the network, but we are not going to wait for it by updating the information simultaneously from the cache. Afterwards, if we get some new content from the network, we can update the data again. Once we get this new information from the network, we can update the view and, as well as that, we can store that information into the cache for the next time. If we can't get the data from the network, then we will stick with what we've got. Taking such approach makes the user satisfied offline, online and even on slow-connections. It will make the user care less about the connectivity.

5.3 IndexDB database

When the user opens an application, we want to show the latest data the device received. Then we make the web-socket connection (a web-socket bypassses both the service worker and the HTTP cache), and we start receiving new data records. When we recieve it, we want to update the application state, taking new data into account. But apart from that, we would like to add new data to already stored one in the cache. We might also think about removing the data, which is too old to keep (depends on the user case).

The database API we can use in this case is IndexedDB. It allows to create multiple databases with a custom name. Each database contains multiple object stores – one for each kind of thing we want to store. An object store contains multiple values - JS objects, strings, numbers, dates, arrays. Items in the object stores can have a separate primary key, which should be unique in the particular store, to identify an object. There are multiple operations that can be done to items in object stores: get, set, add, remove, iterate. All read or write operations in IndexedDB must be a part of a transaction: this means that if we create a transaction for a series of steps and one of those fails – none of them are going to be applied. The browser support of IndexedDB is good, because every major browser supports it.

In case the user wants to make a change to the data, there are two cases to handle.

- The request to the server is succesfull: in this case, the request would end up changing the data on the server's side, while the client will just update its own cache once it gets a response from the server.

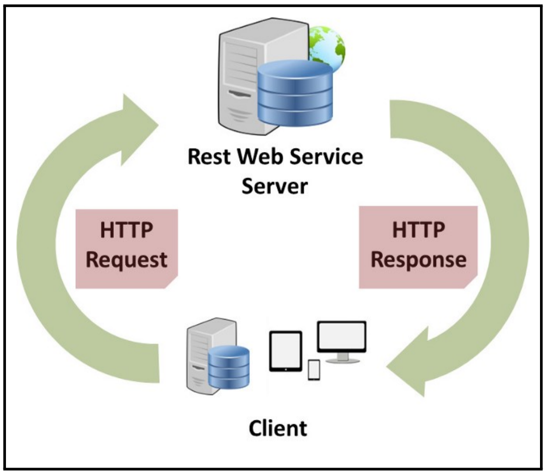
- The request to the server is not succesfull: here, it is going to be a bit more interesting. We will have to wait for an error message and store the data in a temporary database for the updates that are not sent to the server's side just yet. Afterwards, I think it would be logical to have a timer, which will check the connection with the server. Once it is back, every transaction again is going to be sent to the server. After all the data is sent, this temporary database can be cleaned.

Several problems could arise, however, which we will adrress in the architecture chapter.

6 Implementation

In this chapter, we are going to explain the major problems that we encountered during the implementation phase.

Addressing the client, we will name the frameworks and techniques that we are going to use in the implementation phase. Concerning the server, which is going to provide the RESTful interface, though, we will cover the use cases.



Source: https://cdn-images-1.medium.com/max/660/1*EbBD6IXvf3o-YegUvRB_IA.jpeg

Figure 6.1: A look at the server-client architecture with a RESTful interface.

6.1 Optimization

7 Evaluation

Evaluation

This chapter is needed, when results from the previous chapters need to be systematically discussed. This is the case, when an implementation needs to be assessed, or the results of a case study or an experiment need to be interpreted.

8 Related Work

In this chapter, we are going to mention some of the researches, which influenced our work.

8.1 SwiftCloud: a transactional system that brings geo-replication to the client

Geo-replication of data into several data centers (DC) across the world is used in cloud platforms in order to improve availability and latency[15]. This goal could be achieved even to a greater extent by storing some part of the data or even by replicating all of it at client machines. Thus, caching could be useful in terms of increasing availability of systems.

One of such systems that integrates client- and server-side storage is called SwiftCloud, where the idea is to cache a subset of the objects from the DCs, and if the appropriate objects are in cache, then responsiveness is improved and the operation without an internet connection is possible[16]. The authors of the SwiftCloud state that it improves latency and throughput if it is compared to general geo-replication techniques. That is possible due to availability during faults thanks to automatic switch of DCs when the current one does not respond. Apart from that, the system also maintains consistency guarantees [7].

8.2 Legion: a framework, which enriches web applications

A framework named Legion shows another interesting approach how to address availability and scalability issues by using cache at the client-side. The idea is to avoid a concept of a centralized infrastructure for mediating user interactions, as it causes unnecessarily high latency and hinders fault-tolerance and scalability[17]. As an alternative, authors of Legion suggest client web applications to securely replicate data from servers and afterwards synchonize the data among the clients. This change makes the system less dependent on the server and, moreover, it reduces the latency of interactions among the clients. The guarantee of convergence between all of the replicas is possible due to CRDTs, introduced in **Chapter 2**.

8.3 Developing web and mobile applications with offline usage experience

Nowadays, applications with offline experience are becoming extremely popular. In this thesis, in **Chapter 5** we presented an approach of developing web applications with a help of service workers and background synchonization (they also go by name of Progressive Web Applications [PWA]). It is a universal approach to create a cross-platform application that would work in web and on mobile devices. However, there are also other ways how the offline experience could be achieved. Sometimes, the process could become easier if specifc frameworks are used. In the following subsections, we are going to describe tools and techniques that are useful in this context.

Polymer App Toolbox

Polymer is a JavaScript library from Google, which helps to build web applications with the use of Web Components. The former concept represents a set of web platform APIs, which allow to create custom HTML tags to use in web pages[18]. As web components are based on the latest web standards and it eases the process of development. Polymer App Toolbox, in its turn, provides a collection of components to build PWAs with Polymer. However, the support of offline-experience is possible yet again due to Service Workers[19], which repeats the solution used in this thesis. Currently, among the users of Polymer, apart from Google, there are such giants as Electronic Arts, IBM and Coca-Cola[20].

HTML5 Specification

The specification of HTML5 contains some features that offer a possibility to build web applications that work offline. The solution to address this problem is to use SQL-based database API in order to be able to store data locally, and to use an offline application HTTP cache to make sure about the availability of the application when there is no internet connection.

The latter makes possible the following advantages: offline browsing, flexibility and fast load of resources from the hard drive[21]. However, from 2015 the application cache is considered to be deprecated and is recommended to be avoided[22] in favour of Service Workers. Apart from that, HTML5 also lets the developer to check in which stage the user's navigator is, by simply checking the attribute *navigator.onLine*[23]. Offline and online events make changes to this attribute, which reflects the status of current connection. The advantages of HTML5 approach is the reusability of code, shorter development and testing times and universal support of mobile platforms. However, if compared to the mobile applications written in native languages, HTML5 web applications could be inferior to them.

Hoodie

Hoodie is a framework, which eases the process of developing web and iOS applications. We will not cover all the features it offers and only stop on it providing offline experience for applications that are developed using Hoodie. The documentation states that framework is offline-first[24], which means that all the data is fistly stored locally and could be accessed without the internet connection. This is possible due to PouchDB database, which performs this work in the background[25]. We will explore the process of how PouchDB works below.

localForage

localForage is a JavaScript library, which provides the possibility to improve the offline-experience of web applications in terms of storing data on the client-side. It uses IndexedDB, localStorage or WebSQL with a simple API. localForage sits on top of datastore layer and provides a range of methods to control the data. One of the useful benefits of it is that the data is not required to be explicitly converted into JSON format, as localForage does that automatically[26].

PouchDB and CouchDB

PouchDB represents an open-source JavaScript database, which is designed to build applications, which work well online and offline. To put it in a nutshell, PouchDB enables web applications to store the data locally, and, apart from that, eases the process of synchonization with CouchDB-compatibale servers. CouchDB, in its turn, is a database that is supported by a replication approach, which allows to synchonize two or more servers, based on CouchDB. As there is a replication approach, it has its own way of dealing with conflicts, which we explain next by following the description of the protocol offered by Lehnardt [27]. For every item stored in CouchDB, the database will add two extra properties: *_id* and *_rev*, which can be seen at **Listing** 8.1 that shows the result of getting an element named *document* previously stored in the database.

Listing 8.1 A typical result of retrieving the item *document* stored in CouchDB.

```
{"_id":"document","_rev":"1-23202479633c2b380f79507a776743d5"
  ,"a":1}
```

As you can see, the *_id* represents the name of the item, which is a custom name set by the user, while *_rev* represents the hash value, associated with the content. Afterwards, whenever we want to change the item *document*, the same *_id* and *_rev* should be used. For example, to change the value of *document* by adding *b* into it, a simple PUT HTTP-request should be sent, as **Listing** 8.2 shows.

Listing 8.2 Updating the value of item *document* by adding *b* into it.

```
PUT /database/document
{"_id":"document","_rev":"1-23202479633c2b380f79507a776743d5"
  ,"a":1, "b":2}
```

The next retrieval of the *document* will get us the result shown at **Listing** 8.3.

Listing 8.3 The result of requesting the updated version of *document*

```
GET /database/document
{"_id":"document","_rev":"2-c5242a69558bf0c24dda59b585d1a52b"
    ,"a":1, "b":2}
```

As you can see, the *_rev* property got updated. In case the wrong revision is sent to update the document, the database will respond with an error.
However, as you might have guessed, it is much more interesting, when you have more than one server. Let us assume that now we have two CouchDB servers. Imageine we try to update the *document* once again with the values shown at **Listing** 8.4.

Listing 8.4 Updating the value of item *document* by adding *d* into it.

```
PUT /database/document
{"_id":"document","_rev":"2-c5242a69558bf0c24dda59b585d1a52b"
    ,"a":1, "b":2, "d":4}
```

For example, let us say that it gets written to the first CouchDB server, which gives the response shown at **Listing** 8.5.

Listing 8.5 The result of requesting the *document* from CouchDB-1.

```
GET /database/document
{"_id":"document","_rev":"3-2235fd4815b81b2da1b84159aba4006e"
    , "a":1, "b":2, "d":4}
```

But the second CouchDB server still has the old data, as you can see at **Listing** 8.6.

Listing 8.6 The result of requesting the *document* from CouchDB-2.

```
GET /database/document
{"_id":"document","_rev":"2-c5242a69558bf0c24dda59b585d1a52b"
    ,"a":1, "b":2}
```

Ideally, the replication happens fastly and both databases will synchonize and reach the same state. However, sometimes it might take a while. And if a client tries to update the document yet another time, there is a possibility that the update will go to the second server, which will reject the update, as *_rev* does not match any more.

Listing 8.7 Updating the value of item *document* by adding element *e* and removing previously added element *d*

```
PUT /database/document
{"_id":"document","_rev":"3-2235fd4815b81b2da1b84159aba4006e"
    , "a":1, "b":2, "e":5}
```

As we mentioned ealier, the response of the second CouchDB server for the operation shown at **Listing** 8.7 will look like the one at **Listing** 8.8.

Listing 8.8 The demonstration of a conflict situation happening, when the *_rev* of sent operation and the one at the server do not match.

```
{"error":"conflict","reason":"Document update conflict."}
```

However, there is another option to perform this update. We might have a strategy of getting the latest *_rev* first, which was "*2-c5242a69558bf0c24dda59b585d1a52b*" at the second CouchDB server, and only then applying the update. So, the operation will look as at **Listing** 8.9.

Listing 8.9 Updating the value of item *document* by adding element *e* and removing previously added element *d* after receiving the new *_rev* from the second CouchDB server.

```
PUT /database/document
{"_id":"document","_rev":"2-c5242a69558bf0c24dda59b585d1a52b"
    , "a":1, "b":2, "e":5}
```

In case this PUT request goes to the second CouchDB server, then this operation will be successful and now both servers will have different states, which creates a conflict situation. However, it is still an undesirable situation. Thus, there are the limitations that should be given a thought in the development process, which indeed make the process of creating the product based on CouchDB more complicated:

- Making a change, do not request multiple GETs and POSTs
- Do not update the _rev locally in the client without getting new data from the server before that

Realm Mobile

This is a framework, which makes an integration of a client-side database for iOS and Android with a server-side, which offers the following features: real-time synchronization, conflict resolution and event handling. This framework eases the process of developing applications with offline experience. The concept we are interested here is conflict-handling. In AntidoteDB, for this purpose, CRDTs are used. Realm Mobile maintains a good user experience in offline mode due to the rules they described for conflict resolution. As they state, "at a very high level the rules are as follows:

- **Deletes always win.** If one side deletes an object it will always stay deleted, even if the other side has made changes to it later on.
- **Last update wins.** If two sides update the same property, the value will end up as the last updated.
- **Inserts in lists are ordered by time.** If two items are inserted at the same position, the item that was inserted first will end up before the other item. This means that if both sides append items to the end of a list they will end up in order of insertion time."[28]

The authors of the framework state that “strong eventual consistency” guarantees are reached[28] with the above mentioned approach. Though, such way of conflict handling is not as flexible as CRDTs and has to be taken into account by the programmer at the stage of the development.

9 Conclusion

Conclusion

This is always the first chapter of the thesis. The chapter should be short (up to 5 pages). The chapter should feature sections as follows (where applicable): o Summary (Summarize this work in an insightful manner, assuming that the reader has seen the rest.) o Limitations or threats to validity (Point out the limitations of this work. In the case of empirical research, discuss threats to validity in a systematic manner.) o Future work (Provide insightful advice on where this research should be taken next.)

9.1 Summary

9.2 Future Work

- Automatic synchonization of updates like without pressing any buttons;
- The thing that was mentioned in the design chapter about the state-based approach;

Idempotence of updates

- wrote about this part in the notes to the architecture pdf

In case a client is sending an update to the server and does not receive an acknowledgement, what could have happened?

There are possibilities:

- The update was applied on the server, but the connection failed when the acknowledgement was about to be sent;
- The update was not applied on the server and the client did not receive an acknowledgement;

However, the client does not know which of these situations happened. Therefore, we have to find a general solution for such kind of behaviour.

One of the solutions could be the following: it does not matter, whether the update was applied or not. A client will just send the update again, till it does not recieve an acknowledgement, regardless of what happens on the server's side.

The other one is: the client sends an update to the server, which should send a message back that it received an update. Afterwards, a client removes this update from the temporary database and sends back a message to the server that it is possible to apply the update.

However, the implementation of the solution for this problem is beyond the scope of this Master thesis. Therefore, these thoughts are going to be included in a section, where future impovements will be discussed.

List of Figures

List of Tables

List of Code

Listings

Bibliography

[1] M. Tamer zsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2011. ISBN 1441988335, 9781441988331.

[2] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994. ISSN 0734-2071. doi: 10.1145/176575.176576. URL http://doi.acm.org/10.1145/176575.176576.

[3] Irene Y. Zhang. Consistency should be more consistent! https://irenezhang.net/blog/2015/02/01/consistency.html, 2015 (accessed October 23, 2018).

[4] Marc Shapiro. Strong eventual consistency and conflict-free replicated data types. https://www.microsoft.com/en-us/research/video/strong-eventual-consistency-and-conflict-free-replicated-data-types/, 2011 (accessed October 23, 2018).

[5] S. Gilbert and N. Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, Feb 2012. ISSN 0018-9162. doi: 10.1109/MC.2011.389.

[6] Santiago J. Castiñeira and Annette Bieniusa. Collaborative offline web applications using conflict-free replicated data types. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, pages 5:1–5:4, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3537-9. doi: 10.1145/2745947.2745952. URL http://doi.acm.org/10.1145/2745947.2745952.

[7] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, 2015. ISBN 978-1-4503-3618-5. doi: 10.1145/2814576.2814733. URL http://doi.acm.org/10.1145/2814576.2814733.

[8] Antidotedb. https://www.antidotedb.eu/, 2018 (accessed September 25, 2018).

[9] Annette Bieniusa Deepthi Devaki Akkoorath. Antidote: the highly-available geo-replicated database with strongest guarantees. *SyncFree Technology White Paper*, 2016.

[10] Butler W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, 1981. ISBN 3-540-10571-9. URL http://dl.acm.org/citation.cfm?id=647434.726386.

[11] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguica, and Marc Shapiro. Cure: Strong Semantics Meets High Availability and Low Latency. *Proceedings - International Conference on Distributed Computing Systems*, 2016. ISSN 1063-6927. doi: 10.1109/ICDCS.2016.98.

[12] Marc Shapiro, Nuno Pregui, Carlos Baquero, and Marek Zawirski. Replicated Data Types: A comprehensive study of Convergent and Commutative Replicated Data Types. 2011. URL http://hal.inria.fr/inria-00555588.

[13] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. *Conflict-Free Replicated Data Types CRDTs*. Springer International Publishing, 2018. ISBN 978-3-319-63962-8. doi: 10.1007/978-3-319-63962-8_185-1. URL https://doi.org/10.1007/978-3-319-63962-8_185-1.

[14] Matt Gaunt. Service workers: an introduction. https://developers.google.com/web/fundamentals/primers/service-workers/, 2018 (accessed September 14, 2018).

[15] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043592. URL http://doi.acm.org/10.1145/2043556.2043592.

[16] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, SRDSW '14, 2014. ISBN 978-1-4799-7361-3. doi: 10.1109/SRDSW.2014.33. URL https://doi.org/10.1109/SRDSW.2014.33.

[17] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching Internet Services with Peer-to-Peer Interactions. *Proceedings of the 26th International Conference on World Wide Web*, 2017. doi: 10.1145/3038912.3052673.

[18] What are web components? https://www.webcomponents.org/introduction#what-are-web-components-, 2018 (accessed October 29, 2018).

[19] Offline caching with service worker precache. https://www.polymer-project.org/1.0/toolbox/service-worker, 2016 (accessed October 29, 2018).

[20] Who's using polymer? https://github.com/Polymer/polymer/wiki/Who's-using-Polymer%3F, 2018 (accessed October 29, 2018).

[21] Eric Bidelman. A beginner's guide to using the application cache. https://www.html5rocks.com/en/tutorials/appcache/beginner/, 2018 (accessed October 30, 2018).

[22] Using the application cache. https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache, 2018 (accessed October 30, 2018).

[23] Anne van Kesteren. Offline web applications. https://www.w3.org/TR/offline-webapps/, 2008 (accessed October 29, 2018).

[24] hood.ie intro. http://hood.ie/intro/, accessed November 4, 2018.

[25] Jake Peyser. Intro to hoodie and react. https://css-tricks.com/intro-hoodie-react/, 2017 (accessed November 4, 2018).

[26] Matt West. Using localforage for offline data storage. https://blog.teamtreehouse.com/using-localforage-offline-data-storage, 2013 (accessed November 4, 2018).

[27] Jan Lehnardt. Understanding couchdb conflicts. https://writing.jan.io/2013/12/19/understanding-couchdb-conflicts.html, 2013 (accessed November 4, 2018).

[28] Conflict resolution. https://docs.realm.io/platform/self-hosted/customize/conflict-resolution, 2018 (accessed October 30, 2018).