atabibliography/references

Department of Computer Science

University of Kaiserslautern

Master Thesis

Offline caching in web applications for AntidoteDB

Server Khalilov

contact@serverkhalilov.com

University of Kaiserslautern
Department of Computer Science
Software Engineering

Leader:
Prof. Dr. Arnd Poetzsch-Heffter

Supervisor:
Dr. rer. nat. Annette Bieniusa

Zusammenfassung

Zusammenfassung auf deutsch

Abstract

Despite advancements in connectivity for user devices by service providers, such devices are subject to periods of disconnection. In order for the user to be able to interact with the application during periods of disconnectivity, the application must store its data on the client machine. Besides that, updates need to be maintained and delivered to the server once the connection is reestablished. The demand for offline support has led to many ad-hoc solutions that often do not provide well-defined consistency guarantees.

In this thesis, we developed a WebCure, a framework for partial replication of data at client-side in web applications. It consists of a client-side data store that maintains both the data that has been received from the cloud storage server and the updates that have been executed by the client but have not been delivered to the cloud store, yet. A service worker acts as a proxy on client-side. While the client is offline, it forwards the requests to the client-side database. Finally, a cloud storage server maintains data shared between different clients. For this purpose, we chose AntidoteDB as the cloud storage server since it provides with conflict-free replicated data types (CRDTs) a well-defined semantics for concurrent updates. Updates that are executed while a client is offline are concurrent with all other updates happening between the last retrieval from the cloud storage and the next connection and synchronisation of the client. We developed the algorithms for WebCure and implemented a stable prototype of the framework. To evaluate its feasibility and performance, we additionally ported a calendar application that allows now for offline operations.

Acknowledgement

I would like to take this opportunity and express my biggest gratitude to 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 Annette for having the door to her office always open for a discussion, whenever I needed it. I am happy I had a chance to work with her on multiple occasions: firstly, during the time of the lecture on Compiler and Language-Processing Tools; then, while being a research assistant in EU Project "Lightkone"; and, finally, by writing a Master's thesis under her guidance. She was always patient and continuously motivated me to strive for better results. 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** and **sister**. 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 introductory 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 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 – WebCure.

1.2 Research questions

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

- *RQ1.* What could be a scalable solution for the transmission of Conflict-Free Replicated Datatypes data between a client and AntidoteDB-based server?
- *RQ2.* What are the methods and technics available to implement web applications that would be able to work offline and in the conditions of poor network connections?

1.3 Structure of Thesis

In this section, we are going to give a summary of the structure of this thesis. We started with the introduction in the **Chapter 1**, where we discuss the motivation of the thesis in the **Section 1.1** and research questions

in the **Section** 1.2. Then, in the **Chapter** 2, we are going to talk about the fundamentals required to comprehend the idea of the paper: in the **Section** 2.1 we are discussing the main theoretical concepts related to the distributed databases, while in the **Section** 2.2 we introduce the concepts of AntidoteDB, and in the **Section** 2.3 we cover Conflict-Free Replicated Datatypes in general. Additionaly, there we describe the datatypes we used in our work. Afterwards, in **Section** 3.1 and **Section** 3.2 we will talk about the requirements and assumptions we make for the system we are going to design. Next, in the **Section** 3.3 we will show in details how we came up with the protocol of the system. Having done that, in the **Chapter** 5 we will go in details through the implementation of the system, starting discussing the system's main components in the **Section** 5.1. Furthemore, in the **Chapter** 6 we will cover the topic on how we made an evaluation of the system, where we will also show WebCure running in a Calendar application. Finally, in the **Chapter** 7 we discuss other methods and technics that inspired us for this work. Then, in **Chapter** 8 we will summarize the work of this thesis and in the **Section** 8.2 we will talk about any future improvements that can be done.

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"[**?**]. 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 [**?**]. *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[**?**]. Linearizability guarantees that the same operations are applied in the same order to every replica of the data item[**?**]. Serializability is a guarantee about transactions, that they are executed serially (i.e. without overlapping in time) on every set of the data items[**?**]. 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 WebCure.

As stated by **?**], "*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[**?**], 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[**?**]. In this thesis, we will stick with partial *causal consistency*. As it is stated in **?**], 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 respects the cau-

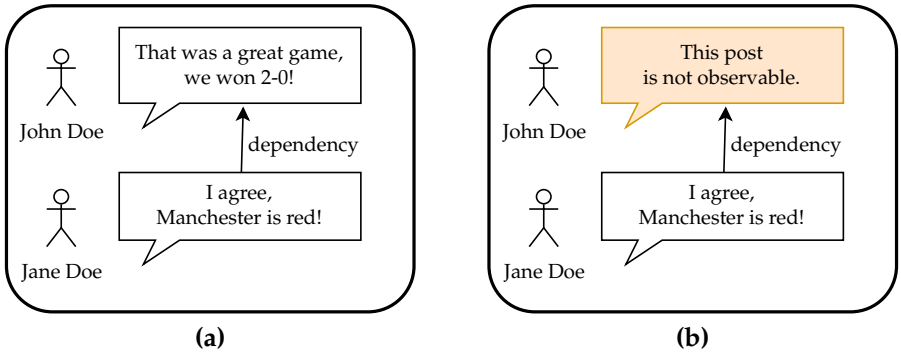


Figure 2.1: An example of how a causally consistent behaviour (a) and the one that is not (b) could work in a social network.

sality 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** we 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 WebCure belongs to the database called AntidoteDB[**?**]. It helps programmers to write correct applications and has the same performance and horizontal scalability as AP / NoSQL[**?**], 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[**? ?**];
- delivers updates in a causal order and merges concurrent operations;

Merging concurrent operations is possible because of Conflict-Free Replicated Datatypes (CRDTs) [**?**], 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[**?**]. We will cover the topic of CRDTs later in this chapter.

Apart from that, to replicate the data AntidoteDB implements the *Cure*[**?**] 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*[**?**].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 (CRDTs)

As it is stated in the work of **?**], 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[**?**]. 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

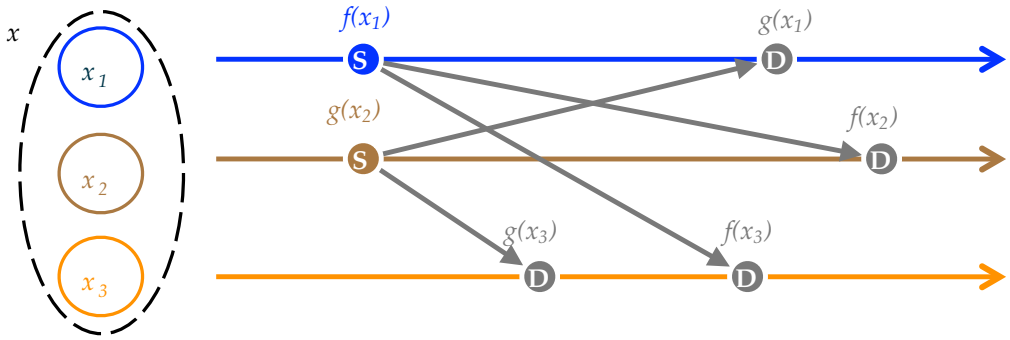


Figure 2.2: Operation-based approach[**?**]. «S» stands for source replicas and «D» for downstream replicas.

In this thesis, we are going to use the operation-based replication approach, where replicas converge by propagating operations to every other replica[**?**]. 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**, we 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 every update eventually is going to appear at every other replica in the system. We 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

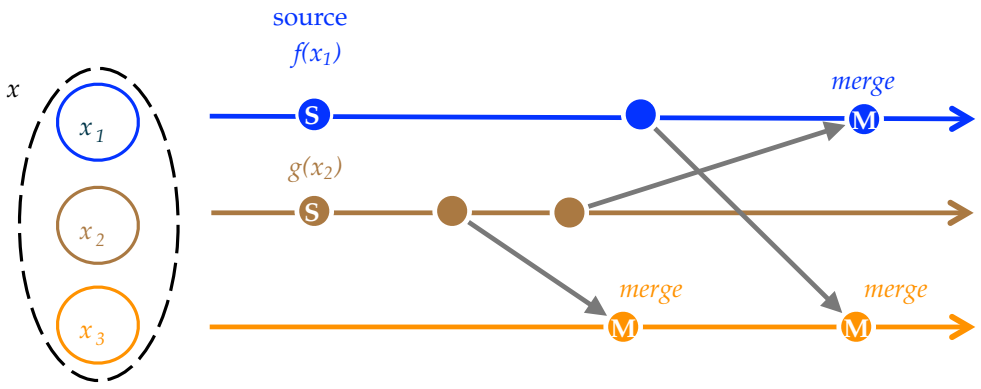


Figure 2.3: State-based approach[**?**]. «S» stands for source replicas and «M» for merging stages.

replicas before sending a new full state for a follow-up merges to other replicas in the system.

There are different types of CRDTs, however, we will consider three of them: counters, sets and multi-value registers. We will give a brief description to each of the mentioned datatypes below.

Counter

This is a datatype, which keeps track on its state, which is an integer number. The value of the Counter could be modified by the operations *inc* and *dec* that increases or decreases the state by one unit, accordingly[**?**]. The concurrency semantics for this datatype is that a final state of the object reflects all the performed operations on it. In other words, to calculate the state of the Counter, it is needed to count the number of increments and substract the number of decrements.

Add-wins Set

Add-wins Set[1] datatype represents a collection of objects with a specific handling of concurrent updates performed over them. In case of concurrent updates on the same object, *add* operations in Set win against *remove* operations. If, for example, there is an empty set and two concurrent operations applied to it – *add a* and *remove a*, then the result is going to be *{a}*,

[1]for the simplicity of reading, later it goes as Set

as *add* operation wins. A *remove* operation will "overwrite" an *add* operation only when it happens after it[**?**].

Multi-value Register

This datatype maintains a value and provides a *write* operation of updating that value. The interesting part about Multi-value Registers is their concurrency semantics. In case of two or more updates happening at the same time, all values are kept. Thus, the state of the register will consist of all the concurrently written updates for a further processing. However, any additional single *write* operation will overwrite the previous state of the register, even if it consisted of multiple values[**?**].

3 Design

In this chapter, we are going to, firstly, introduce the requirements and assumptions we make for the design of WebCure. Having set them, afterwards, we will in detail describe the design of the system.

3.1 Requirements

We listed the functional requirements of WebCure in **Table 3.1** and non-functional requirements in **Table 3.2**.

First of all, looking at the first three functional requirements, *R1–R3*, we see that for the implementation were selected such CRDTs as Counter, Set and Multi-Value Register. The reason for that is that these data types cover the most operations on CRDTs and, if our design works for them, it will work for the rest as well. The requirement *R4* makes the user able to get the data from the server at different timestamps and compare it. Next, talking about the requirement *R5*, even though it is possible to request data at different timestamps, it is vital for the client to store in cache only the latest available at the server data. The requirement *R6* is due to one of the limitations AntidoteDB possesses, and is related to one of the assumptions we make in the following section. Requirements *R7–R9* specify what kind of actions the user is able to perform on a client when offline, as well as their synchronisation with the server. Finally, the requirements *R10* and *R11* guarantee that the changes performed offline at the client, as well as changes at the server side, are synchronised in a way satisfying causal consistency model.

Next, there are non-functional requirements: *NFR1* and *NFR2*. Basically, both of them support our claim that a client is able to work offline, as well as under uncertain network conditions.

3.2 Assumptions

In this section, we are going to give a list of assumptions we make for WebCure.

Table 3.1: Functional requirements.

R1	Retrieval, increment and decrement of the Counter CRDT should be possible.
R2	Retrieval, adding and removing elements from the Set CRDT should be possible.
R3	Retrieval, assigning and resetting the Multi-Value Register CRDT should be possible.
R4	Retrieval elements of any supported CRDTs should be possible according to the passed timestamp.
R5	The client should cache only the latest data available at server's side.
R6	It should not be possible to create elements of different CRDTs with the same name (due to limitations of AntidoteDB).
R7	When offline, it should be possible to make read/write operations on supported CRDTs.
R8	The user should be able to remove from the client any stored data element.
R9	Any operations performed offline, once the connection is restored, should be sent to the server immediately.
R10	The execution model of offline operations at the client should be sequential (updates are ordered).
R11	When the connection is reestablished after having data changes in offline mode, the client storage should be updated appropriately (with a consideration of the client's offline changes and possible changes on the server).

Table 3.2: Non-functional requirements.

NFR1	The system should be available online and offline (except for the functionality with timestamp-related updates).
NFR2	The system should be available with a poor internet connection.

1. **Timestamps**. Firstly, the database storage used for the server's side should have the concept of timestamps (like in AntidoteDB, described in **Section 2.2**), in order for the protocol we are going to describe in the **Section 3.3** to work correctly.

2. **Cache is persistent**. For WebCure to work online and, especially, offline, we believe that the browser's cache is safe from automatic clearing. Contrarily, if the cache could be cleared automatically depending on the browser's behaviour, it makes it impossible to support the claim that the application can work offline. To guarantee this assumption, a Persistent Storage API described in **Section 4.4** can be used.

3. **Name duplicates**. We limit the creation of different CRDT elements with the same name in the system due to limitations of AntidoteDB, as the requirement *R7* describes it in the **Table 3.1**. As AntidoteDB is in the process of ongoing development, currently the database crashes when there is an attempt to create elements of different CRDTs with the same name. Thus, this condition has to be fulfilled.

4. **Server's database is always on**. We assume that the server's database is not going to be reset and lose all its data. As the client entirely relies on the server's storage for the synchronisation and only sends back operations performed offline on client's side, it will be impossible to restore the server's database from the client's storage, even if it was up-to-date before the server's data loss. With additional changes to the current protocol, it might be possible, though, but that is not the topic we cover in this thesis. However, even in such a situation, the client will be able to continue the offline work.

As we specified the requirements, we can go further into and design the protocol of the system.

3.3 Protocol

The fundamental part of WebCure will be its protocol design. We are going to describe it in an event-based way in the form of pseudo-code in the following sections.

3.3.1 Data transmission

As we already remember from the **Chapter 2**, because AntidoteDB is using CRDT datatypes, the following options are possible to update the database: state-based and operation-based. This thesis will consider only the operation-based approach, as it has such an advantage over the state-based approach as less transferred data in most situations. 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, Sets and Multi-Value Registers, to which the reader was introduced in the **Section 2.3**.

3.3.2 Description

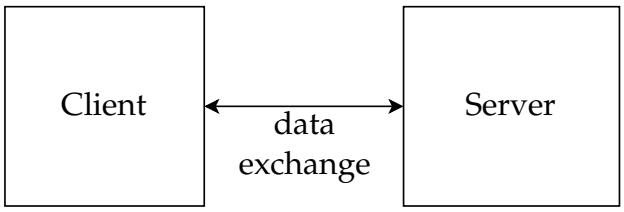


Figure 3.1: An overview of the communication protocol.

Firstly, as we would like to focus on the communication part between a server and a client, let us for now keep both of them as black boxes[2], as they are represented at **Figure 3.1**. Next, we will go through different stages of their communication and describe, how we handled these processes.

Graphics notations

Let us explain the notations, which are going to be used for a further protocol description. At **Figure 3.2 (a)**, we can see a notation for the timeline. Timelines will be used for the matter of showing the sequence of events happening. At **Figure 3.2 (b)**, the arrow shows the transmission of data between a subsystem A and subsystem B, as well as its direction and a

[2]In software engineering, a black box is a system, which can be viewed in terms of its inputs and outputs, without the understanding of its internal workings.[**?**]

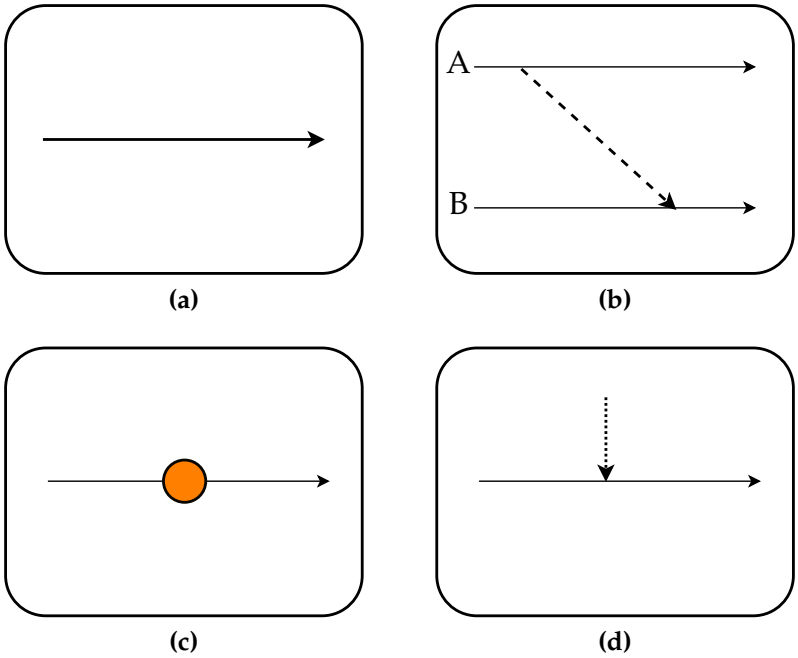


Figure 3.2: An overview of notations used in the following chapters for the protocol explanation.

command. **Figure 3.2 (c)** represents the state of the data on a system's side, while **Figure 3.2 (d)** points to a timestamp, at which an operation that changes the system's storage was applied.

Next, as we already mentioned, we will explain the protocol in an event-based way.

A client receives an update from the server

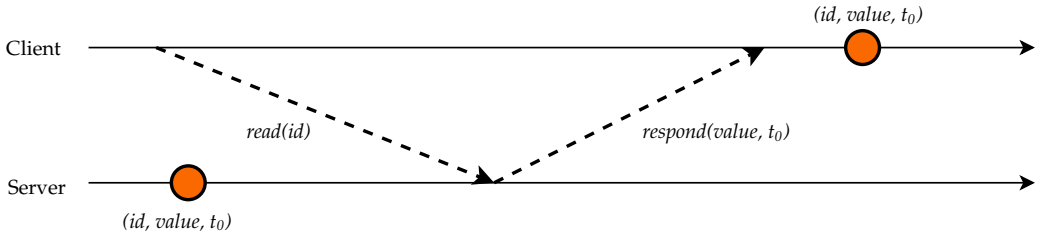


Figure 3.3: The communication between a client and a server for the read function.

Let us assume that a client initiates its work with an empty storage. Then, a user might want to request the actual data from the server. In this case, as can be seen at **Figure 3.3**, a user has to pass to the server an *id* of the data to read. If the request is successful, the server is going to respond with a *value* for the requested *id* and the timestamp of the last write – t_0, so the client will store this information in its own storage.

Listing 3.1 Pseudocode for requesting the data: client.

```
// Read function that pulls database changes
// @param id: the id of the object, for which the update was
    requested;

function read(id) {
  GetHttpRequest( // send an http-request to get the data
      from the server by id
    id,
    function onSuccess(value, timestamp) { // get the value
        and timestamp from the server

      // create an object that maintains all necessary data
      var item = {
        id: id, // id of an data item
        operations: [], // operations performed offline
        sentOperations: [], // operations performed offline
            and already sent to the server
        state: value, // a state received from the server
        type: 'set' // a type of CRDT
      };

      storeInCache(item); // cache an item created earleir
      storeInCache(timestamp); // cache the timestamp
    },
    function onFail() {
      getFromCache(id); // get the object from cache by id
    }
  );
}
```

The pseudocode of the logic for this *read* functionality can be seen at the

Listing 3.1. At the *line 5*, an asynchonous function *GetHttpRequest* has two callbacks – *onSuccess*, in case a request proceeds successfully, and *onFail* in case of a failure.

In case of success, the value and the timestamp associated with passed *id* are going to be fetched from the server. After that, at the *line 10* we create an element *item*, which has the following properties: *id* for the id of a data item, *state* for the actual state of the data item on the server, *type* for the type of CRDT, *operations* for the operations performed at the client's side while offline, *sentOperations* for the operations performed at the client's side while offline, but which are already sent to the server. Next, a function *storeInCache* is called twice at the lines *18 and 19*. Firstly, to store in cache an object *item* for the offline use. Secondly, to store in cache a *timestamp* received from the server. A client will always receive either the data associated with the latest timestamp from the server or, if a client chooses to specify the timestamp, it will receive the data associated with that timestamp.

In case of failure, however, the method *getFromCache* is going to be called with an argument *id*, as can be seen at the *line 22*. If *id* exists at client's cache, then it will be returned.

Now, let us say, that after receiving an element *id* from the server, the client wants to change it and send it back to the server.

A client sends an update to the server

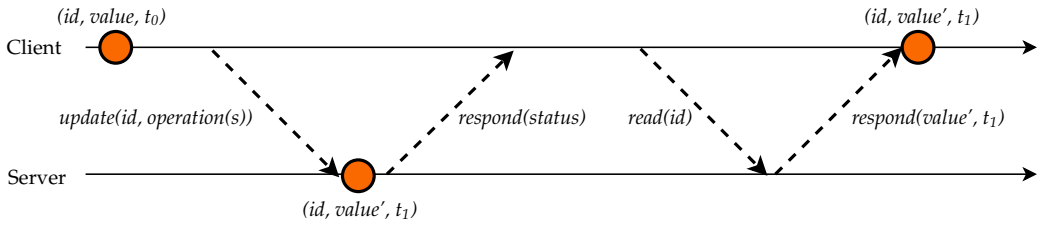


Figure 3.4: The communication between a client and a server for the update function.

Looking at the **Figure 3.4**, in the case of writing the information to the server, a client has to send an *id* with an operation to perform. After that, the server is going to apply the received operation on its side and, in case of success, the new state of the data will receive a timestamp t_1, and an acknowledgement of the successful commit will be sent back to the

client. What happens in case of unsuccessful acknowledgement will be explained below.

So, once the client is notified that the update was applied on the server successfully, a user can get the latest changes to the client's side now. Thus, when the read request for *id* is sent again, the server will send back the new value – *value'* and a new timestamp – t_1, for the element *id*.

Listing 3.2 Pseudocode for making a request to change the data: client.

```
1 // Update function that processes user-made update
2 // @param id: an id for the object that should be updated;
3 // @param op: operation performed on the object for the
      specified id

5 function update(id, op) {
6   PostHttpRequest(
7     // send an http-request to update the data on the server
8     { id, op },
9     function onSuccess(result) {
10      // no actions performed
11    },
12    function onFail() {
13      var item = getFromCache(id); // get the object from
            cache by id
14      item.operations.push(op); // store the operation on a
            client's side in order to try sending it again later
15      storeInCache(item); // cache an updated item
16    }
17  );
18 }
```

However, let us look at the pseudocode placed at the **Listing 3.2**. There we can see the function *update*, which has to have an access to the parameters *id* and *op*. There is an attempt to send the operation *op* for the element *id* to the server, by using *PostHttpRequest*. It is asynchronous and has two callbacks – *onSuccess* and *onFail*, just as before it was explained for the *read* function. If the request succeeds, then the client gets notified about it and no further actions are taken. However, in the case of failure, as we can see at the *line 13*, firstly we are getting the object from the cache by *id*. If it exists, then we add the operation *op* to that object into its property

operations, and, afterwards, store an updated object in cache again using *storeInCache* at the *line 15*. That makes the update available while the user is offline and gives an opportunity to send the operation again when the connection is back.

Next, let us say that a client loses its network connection, so any updates made from that point onwards will be stored locally.

Offline behaviour

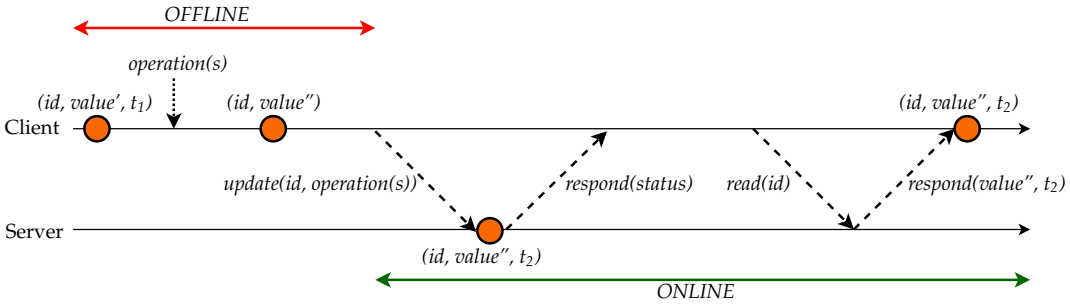


Figure 3.5: The communication between a client and a server while offline with a transition to online.

Have a look at the **Figure 3.5**, where appropriate markings can clearly distinguish periods when the client was offline and online. The client has an element *id* with a value *value'* at timestamp t_1 and then makes a local change applying some operation, which changes the previous value to a new one – *value"*. Pay attention that a new value does not receive a timestamp assigned to it, while locally: to support the causal consistency claims, the responsibility for assigning timestamps should be taken by the server. Then, after some time, the connection gets back, and the client sends an immediate update message to the server with *id* of an element and the applied operation. The server's side, as was already described above, applies that operation and returns an acknowledgement of success. Eventually, the client sends a *read* message and gets back the *value"* as well as the assigned to it timestamp t_2.

Listing 3.3 Pseudocode for sending offline performed operations to the server: client.

```
1 // Update function that processes user-made updates performed
      offline when the connection is restored
2 // @param offlineOperations: an array that contains all the
     operations performed on a client's side offline
3 function synchronize(offlineOperations) {
4   if (ONLINE) {
5     offlineOperations.forEach(operation => {
6       send(operation);
7     });
8   }
9 }
```

The logic described above can be seen at the **Listing 3.3**, which has the function named *synchronize* that should be triggered at the time when the client's side restored a network connection. There, we can see that every operation performed offline is sent once at a time to the server. For causality, the array should be sorted in the order the operations were performed initially.

Now, let us move to the point when more than one client interacts with a server, in order to see how scalable the described protocol is.

Two clients interact with a server.

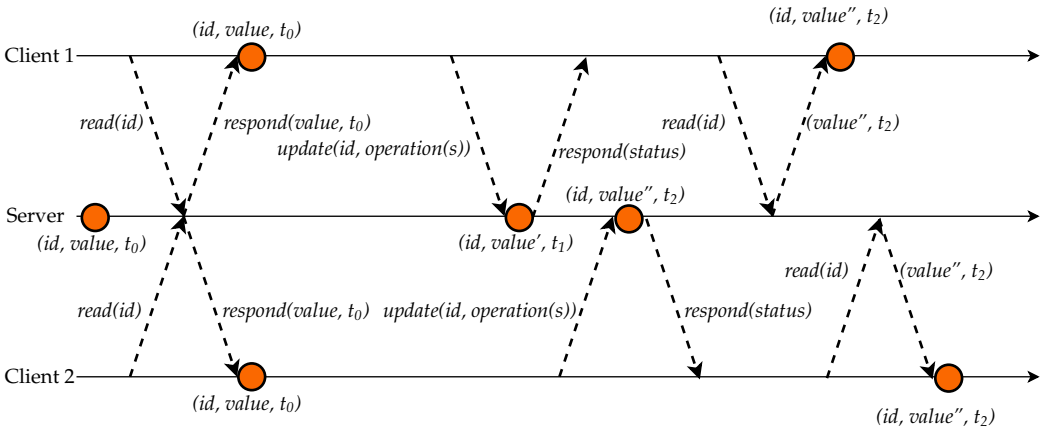


Figure 3.6: The communication between two clients and a server.

Let us assume that, initially, as can be seen at the **Figure 3.6**, the server

has a stored element *(id, value)* at the timestamp t_0. Therefore, when both clients request to read the data from the server, they get that data and store it locally. At the representation above, a *Client 1* is acting first and sends an update to the server changing the value of an element *id* to *value'* at t_1. Observe that *Client 1* does not request the latest data from the server and still only has its local changes. In parallel, a *Client 2* makes the change later at time t_2, and an element *id* is now set to *value''*. Then, both clients request the updated data from the server and both receive the actual value of the element *id* at the timestamp t_2, which is *value''*. We would like to stress the point that all systems - the server and both clients end up having the same data.

Now we would like to give a brief overview of the next two chapters: firstly, in the **Chapter 4** we will give a proper introduction into the technologies we used to implement the described protocol and, then, in the **Chapter 5** we will go through its development.

4 Technologies

In this chapter, we describe the technologies we used to implement the design we introduced in the **Chapter 3**. To give an overview, the implementation of WebCure's design demands the following components:

- A service worker, in order to manage requests coming from the client;
- A Cache API to store HTML, CSS, JavaScript files, and any static files[**?**] to make the application available offline;
- A database to store the data locally;
- Background Sync for deferring the actions conducted offline until the connection is stable;

4.1 Service Worker

Service worker[**?**] is a web worker[3], which is a JavaScript file, which lies in the middle, between the web application and network requests. Service worker runs independently from the web-page and has the following characteristics:

- It does not have access to the DOM[4], however, it has the control over pages (not just a specific one);
- When the application is not running, a service worker still can receive push-notifications from the server[**?**], which let us improve the user experience of the application and makes it "closer" to native mobile applications;

[3]Web Workers make it possible to run a script operation in a background thread separate from the main execution thread of a web application[**?**].

[4]DOM, or **D**ocument **O**bject **M**odel of the page defines HTML elements as objects, as well as their properties, methods, and events.

- It runs over HTTPS, as for the projects using a service worker, the "man-in-the-middle"[5] attacks could represent a real threat;

- It offers a possibility to intercept requests as the browser makes them and has the following options:
  - to let requests go to the network as usual;
  - to skip the network and redirect requests to get the data from the cache;
  - to perform any combination of the above;

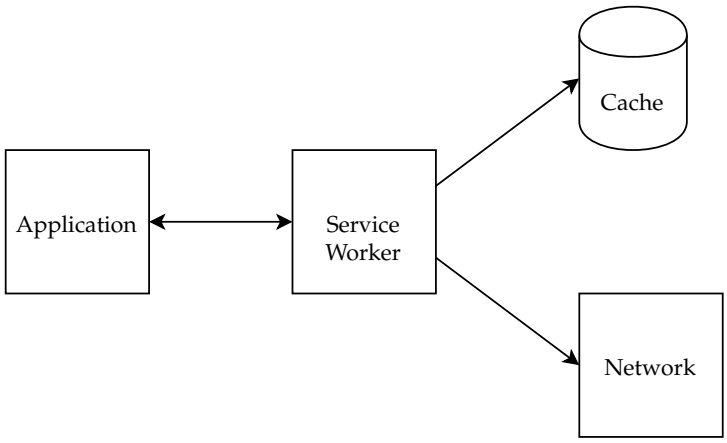


Figure 4.1: An overview of the service worker being able to provide the application experience of both online and offline modes.

We are going to use a service worker in order to intercept the network traffic. For example, as can be seen at **Figure 4.1**, in circumstances when problems are happening with a network when a request is in processing, we will get the data from the cache. At times, when the internet connection could not be established, the cache content can be easily displayed without forcing a user to wait, which dramatically improves performance and user experience. Also, when the internet connection is present, the data can be received from the actual network.

[5]In computer security, this is a type of attack, where a third party secretly alters the communication between two parties, while they believe they are directly "talking" to each other[**?**].

4.1.1 Service worker lifecycle

In this subsection, we will introduce the steps through which a service worker goes. These steps are registration, installation, and activation. All of them we are going to describe below in more details.

Registration

Before one can use service worker features, a developer has to register the corresponding script in the JavaScript code. The registration helps the browser to find a service worker and, afterwards, start its installation in the environment.

Listing 4.1 An example code, which demonstrates how to register a service worker[**?**].

```
1 if ('serviceWorker' in navigator) {
2   navigator.serviceWorker.register('/service-worker.js')
3   .then(function(registration) {
4     console.log('Registration successful, scope is:',
          registration.scope);
5   })
6   .catch(function(error) {
7     console.log('Service worker registration failed, error:',
          error);
8   });
9 }
```

Looking at the *line 1* of code at **Listing 4.1**, we see that, firstly, it is necessary to check, whether the browser supports service workers by observing the property *serviceWorker* of *navigator*[6]. If so, the service worker is going to be registered then, as can be seen at the *line 2*, where the location of the service worker is stated as well. The *navigator.serviceWorker.register* returns a promise, which resolves when the registration was successful. Afterward, the scope of the service worker is logged with *registration.scope*.

Not every browser supports the functionality of service workers. Nowadays, the latest versions of such browsers as Chrome, Edge (partially)

[6]The navigator object contains information about the browser[**?**].

and Firefox support it. However, it is still only partially supported in Opera and Safari, and not at all in the Internet Explorer[**?**] – that is something developers should keep in mind.

The scope of the service workers is quite significant: it defines the paths, from which it can intercept network requests. By default, the scope of the service worker is the location, where the service worker file is stored, including all the sub-directories. For example, if the scope is the root directory, then the service worker is going to regulate the requests for all files at the domain.

Listing 4.2 An example code, which demonstrates how to set a custom scope when registering a service worker[**?**].

```
1 navigator.serviceWorker.register('/service-worker.js', {
2   scope: '/app/'
3 });
```

When registering, it is also possible to use a custom scope. **Listing 4.2** demonstrates that the service worker is going to have a scope of */app/*. It indicates that it will control requests from all the pages like */app/* and deeper. When the service worker is already installed, *navigator.serviceWorker.register* returns the object of the currently active service worker.

Installation

After the registration, the browser might attempt installation of a service worker, if it is considered as a new one, which happens in the following situations:

- when the site does not have a registered service worker yet;
- when there is a difference between the previously installed service worker and the new one;

Installation triggers service worker's *install* event. There is a probability of having a listener for it in order to assign a specific task (depends on the use case), which follows the installation on success. The way to set up this listener is shown at **Listing 4.3**.

Listing 4.3 An example code, which demonstates a listener for the *install* service worker's event[**?**].

```
1 // Listen for install event, set callback
2 self.addEventListener('install', function(event) {
3     // Perform some task
4 });
```

Activation

After the installation, a service worker has to be activated. In case there are open pages, which are controlled by the previous version of a service worker, then the recently installed service worker will be waiting. The activation of a new service worker takes place only when there are no other pages, controlled by the old version of a service worker. That provides a guarantee that only one version of service worker manages the pages of its scope at any time.

Similar to the installation, the activation phase also has its event that gets triggered once the service worker is activating, as we can see at **Listing 4.4**. For example, at this point, it might be a good idea to clean the previously stored old cached data.

Listing 4.4 An example code, which demonstates a listener for the *activation* service worker's event[**?**].

```
1 self.addEventListener('activate', function(event) {
2   // Perform some task
3 });
```

4.2 Cache API

For the best offline experience, the web application should store somewhere HTML, CSS, JavaScript code, as well as images, fonts. There is a place for all of it called Cache API, which we are going to use for Web-

Cure. With Cache API it is possible to store network requests associated with corresponding requests.

Listing 4.5 An example code, which demonstrates how one can create cache storage called *my-cache*[**?**].

```
1 caches.open('my-cache').then((cache) => {
2   // do something with cache...
3 });
```

At **Listing 4.5** we see an example code of how a cache with a name *my-cache* is created. If the operation is performed successfully, then a promise[7] resolves and one is going to get access either to a newly created cache or to the one, which existed before the call of *open* method.

That covers the main functionality of the API. After creating a cache, it is possible to manipulate it in many ways.

Let us now introduce one of the operations – adding elements to the cache. A developer can provide a string for the URL that will be fetched and stored in a cache object. Whenever the data is received from the cache, the browser will return a particular object according to the URL that was stored in the cache. More details on how long the data could be stored in the cache are given in **Section 4.4**.

Apart from adding to the cache object, it is also possible to remove stored URLs, check the current list of cached requests, delete a cache object, and other operations.

4.3 IndexedDB

As the protocol we introduced in the **Chapter 3** clearly describes that there is a necessity to have a client-side storage system, we are going to use for that purpose IndexedDB[**?**]. It is a large-scale, NoSQL database, which lets us store any data in the user's browser. Apart from that, it supports transactions and achieves a high-performance search due to the usage of indexes.

[7] JavaScript Promise is an object, which represents the eventual completion or failure of an asynchronous operation[**?**].

4.3.1 IndexedDB terms

In order to properly understand how IndexedDB works, it is quite useful to understand the concepts that are used in the database. First of all, each *IndexedDB database* contains *Object stores*. Those object stores, in its turn, are similar to tables in traditional databases. Usually, the practice is to have one object store for each type of data. This data could be anything: custom objects, strings, numbers, arrays, and other. It is possible to create more than one database, which could contain various object stores, but normally it is one database per application, which should have one object store for each type of data stored. To expedite the way of identifying objects in object stores, the latter have *primary keys*, which must be unique in the particular object stores. Primary keys are defined by the developer and are very useful regarding searching the data.

All read and write operations in IndexedDB should be wrapped into a *transaction*, which guarantees the database integrity. The critical point is that if one operation within a transaction fails, then none of the other operations are going to be applied.

4.3.2 IndexedDB Promised

As IndexedDB is relatively a new API, it is not supported by all the web browsers yet and, therefore, the support for it should be checked before any further development, as it is shown at **Listing 4.6**. However, all the recent versions of the major web browsers are compatible with it.

Listing 4.6 An example code, which demonstrates how one can check the support for IndexedDB API[**?**].

```
1 if (!('indexedDB' in window)) {
2   console.log('This browser doesn\'t support IndexedDB');
3   return;
4 }
```

Nevertheless, one of the most significant problems with IndexedDB is using it in the development. It has an asynchronous API, which is using events. That makes developers produce a complex code, which is hard to maintain. Therefore, in the design of WebCure, we are going to use a

wrapper over the IndexedDB API – IndexedDB Promised[**?** **?**]. It is a tiny library, written by Jake Archibald from Google, which makes the use of JavaScript promises and simplifies the development process with IndexedDB, while keeping its functionality.

4.4 Persistent Storage

Both Cache API and IndexedDB database mentioned in **Section 4.2** and **Section 4.3** are taking the place at the local machine. However, when the local machine is running out of storage space, user agents will clear it automatically on an LRU policy[8][**?**]. This is not something that suits an application, which promises to provide offline experience. And in the worst case scenario, if the data was not synced with the server, it is going to be lost. The solution for this problem is to use a Persistent Storage API[**?** **?**], which guarantees that cached data is not going to be cleared if the browser comes under pressure.

4.5 Background Sync

The reality is that it is not always possible to be online all the time, even if someone wanted to. Sometimes, there is no network connection at all, or it could be that abysmal that it could be hard to do anything under such conditions. Therefore, the scenario when someone could do his or her work offline and then, when the connection is re-established again, this work will go online, is useful. Nowadays, thanks to *Background Sync*[**?**] from Google it is possible to do in web applications.

Listing 4.7 An example code, which demonstrates how to register a sync (*myFirstSync* here) event for the service worker[**?**].

```
1 // Register the service worker:
2 navigator.serviceWorker.register('/sw.js');

4 // Then later, request a one-off sync:
5 navigator.serviceWorker.ready.then(function(swRegistration) {
```

[8] In the systems with LRU or least recently used caching policy, the least frequently used data will be cleared first.

```
6   return swRegistration.sync.register('myFirstSync');
7 });
```

For this feature to work, the web application has to use a service worker. First of all, it should be registered and, afterwards, a unique tag should be registered as well, which is going to be responsible for the background call of the method. Let us show an example: at **Listing 4.7**, we can see how a synchronization tag *myFirstSync* is registered.

Listing 4.8 An example code, which demonstates that a function *doSomeStuff* called, when the *sync* event happened[**?**].

```
1 self.addEventListener('sync', function(event) {
2   if (event.tag == 'myFirstSync') {
3     event.waitUntil(doSomeStuff());
4   }
5 });
```

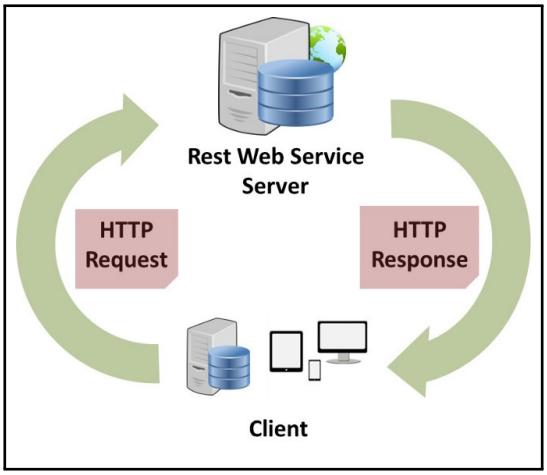
Afterwards, when a page controlled by the service worker is going back online (which is when the user agent has established a network connection[**?**]), a *sync* event is triggered. There, it is possible to perform a distinct action, depending on the registered earlier tag. For instance, at **Listing 4.8** we can see that a function *doSomeStuff* is called once the connection is back, after a *sync* event occurred. This function has to return a promise, which could be successful or not. In the latter case, another sync will be scheduled to try when there is a connectivity[**?**].

Background synchronization is an advantageous feature, which can be used in different scenarios. It could be, for example, sending the e-mails or any other type of messages, after having failed to send it when the connection was poor. However, in our case, in WebCure we are going to use it for sending the operations conducted offline back to the AntidoteDB server for further synchronization. Moreover, if there was a use case with a requirement to get the most recent data from the server after the re-connection to the network, it is possible to implement it using the background synchronization feature.

5 Implementation

In this chapter, we are going to describe the way we solved the obstacles that we faced during the development phase.

In **Chapter 4** we already named the key frameworks and techniques that we used in the implementation for the client. The server is going to provide to the client a RESTful[9] interface for the use cases, which we will describe below.



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

Figure 5.1: A server-client architecture with a RESTful interface.

To demonstrate the functionality of WebCure, we implemented three different types of CRDTs used in AntidoteDB: a Counter, a Set and a Multi-value Register. We support each of these data types on the server and the client as well. We will use a Set CRDT as an example for code listings introduced further.

[9]REST or Representational State Transfer is a set of design principles, which make the network communication more flexible and scalable[**?**].

5.1 Main components of the system

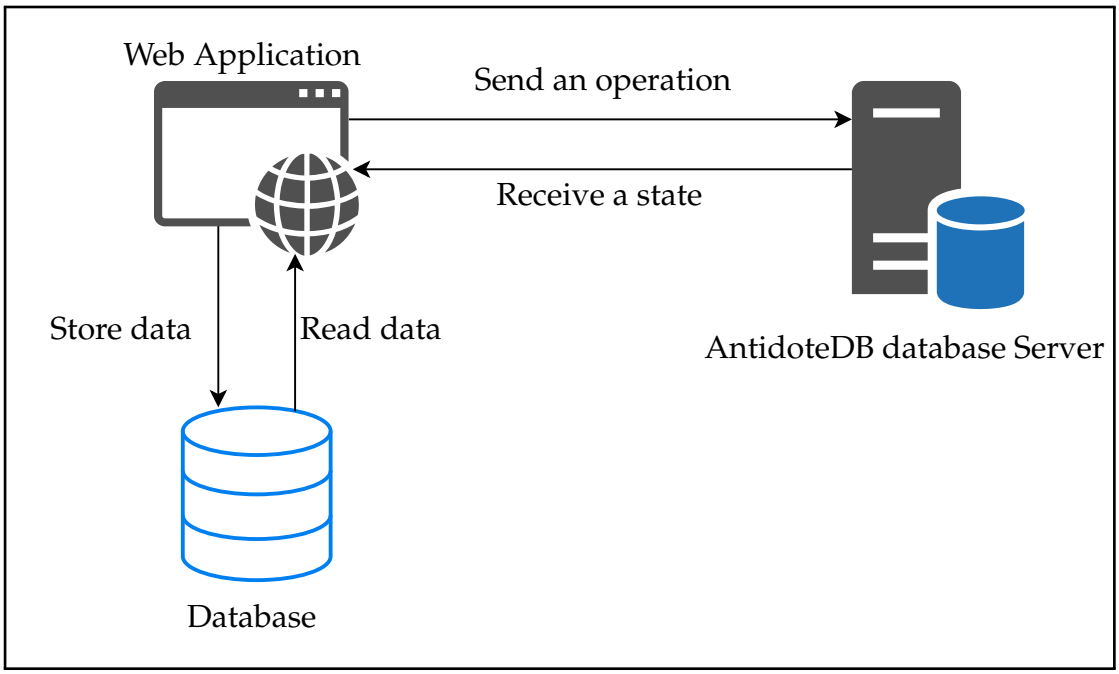


Figure 5.2: A top-level view of the system's design.

In **Chapter 3**, we introduced a client and a server as black boxes, without really explaining their internal structure. However, we have a clear picture of what the system and its main components will look like. As we can see at the **Figure 5.2**, the client part of the system, which we introduced before, contains a web application that has a database on its side for reading and storing the data locally. The server, on the other hand, is controlling an AntidoteDB database, apart from that, can exchange messages with clients. The communication with the server on a client side is controlled by a service worker. It serves as a proxy and gets the data either from cache, or from the server, depending on the use case. Regarding the exchanging of the data, the client can send to the server the operations performed offline, while the server is sending back to the client the current states of CRDT data.

We are going to design WebCure using a 3-tier architecture[**?**]. Its principal distinction is that it has a clear separation between the presentation, application, and data layers, as we can spot in **Figure 5.3**, where we also sketch how each JavaScript file of WebCure relates to these layers. Each

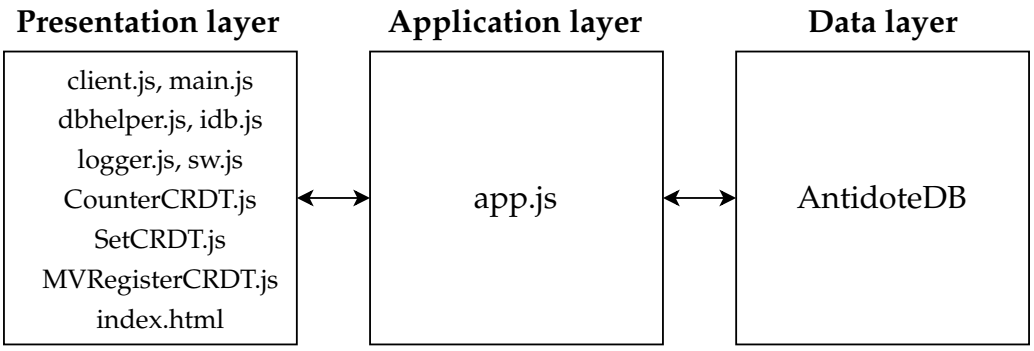


Figure 5.3: 3-Tier architecture.

layer is responsible for its tasks. Let us now briefly characterise them. A *presentation layer* is implemented on a client side and is responsible for presenting the information to the users, while it can also accept requests from them. An *application layer* is based on a server's side, implements the algorithmic part of the system and answers the operations requested by the client. The last one, a *data layer*, manages and implements the data sources of the system. The advantages of a 3-tier architecture style are the scalability and portability it provides. However, a disadvantage could be a communication overhead between the layers[**?**].

In the next subsections, we are going to describe the implementation of the system.

5.1.1 Database

This layer includes an AntidoteDB database, which is used to store and handle CRDT-objects. To be able to work with the database, we are going to use a JavaScript Client for AntidoteDB[10], which suits the stack of the technologies we are using for WebCure. The database itself will be running on a Docker[11] container created from an image of the Antidote data store. For this thesis, however, specific changes were applied to the docker image of Antidote and the JavaScript Client for Antidote as well. In order to read and to apply changes to the data at a specific timestamp, the Antidote image for docker should be running with certification checks disabled, while also there is a flag, which was added to the Antidote JavaScript client to make sure that the client supports the possibility of comitting updates at specific timestamps. Apart from that, the database was

[10]https://antidotedb.github.io/antidote_ts_client/
[11]https://www.docker.com/

also changed to support this functionality.

5.1.2 Server

The server side of WebCure is fundamental, as it provides the interface to the client and manages the communication with the database layer. To implement a server we used a Node.js[12] framework called Express[13]. Taking Set CRDT as an example, we will now discuss the methods the server implements:

- Sending back either the latest state of a Set CRDT or, if a timestamp was provided, then the state at a specific timestamp;
- Adding/removing elements from a Set CRDT;
- Applying a list of operations, which were performed at the client while offline.

Sending the data back

In this section, we will describe how we implemented the functionality of the server to send the data back, as requested by the client.

Listing 5.1 Code for sending back to the client the requested data.

```
1 apiRouter.route('/set/:set_id/timestamp').post(async function
      (req, res, next) {
2       /// ...

4     var setId = req.params.set_id;
5     var timestamp = req.body.timestamp;

7     setTimestamp(timestamp,..);

9     let tx = await atdClient.startTransaction();
10    let set = tx.set(setId);
```

[12]Node.js is a JavaScript run-time environment, which let us execute JavaScript code outside the web browser. More details can be found here: https://nodejs.org/en/.

[13]Express is a Nodej.s framework for web and mobile applications that provides such features as robust routing, HTTP helpers and others. More details can be found here: https://expressjs.com/.

```
11      let val = await set.read();

13      await tx.commit();

15      // ...
16      res.json({
17        status: 'OK',
18        cont: val,
19        lastCommitTimestamp: atdClient.getLastCommitTimestamp()
              .toBase64()
20      });
21    // ...
22  });
```

Have a look at the **Listing 5.1**, where at the *line 1* there is an object *apiRouter* from Express framework introduced above, which adds an HTTP Post route to listen for. As a response to that request, starting from *line 4*, we can observe the logic of the function. There, firstly, we assign the passed parameters to variables. The id of the requested Set CRDT – *req.params.set_id*, is assigned to the variable *setId* and an optional timestamp – *req.body.timestamp*, is assigned to the variable *timetamp*, which is used in the function *setTimestamp* to set the current timestamp of the database. For this operation, the timestamp parameter is optional, though.

At *line 9* we use the asynchronous method *startTransaction* of object *atdClient*, which represents the Antidote JavaScript Client. From that point, a reference of the Antidote Object associated with *setId* is assigned to the variable *set* and afterwards its value is read and assigned to *val*. Then, at the *line 13* the transaction is committed and, afterwards, a JSON object containing the status of the request, the value of requested Set CRDT and the timestamp of the transaction (as the timestamp is an object representing a type of ByteBuffer[14], for a successful transmission in a JSON format, it is converted to Base64[15] at *line 19*. That eases the process of passing the timestamp information in a JSON format.) are sent back to the client.

[14] https://github.com/dcodeIO/bytebuffer.js

[15] Base64 – a group of similar binary-to-text encoding schemes that represent binary data in an ASCII string format by translating it into a radix-64 representation[**?**].

Adding / removing elements

As operations of adding and removing elements from the Set CRDT do not differ regarding implementation, we will use the operation of adding elements as an example for further explanation.

Listing 5.2 Code for applying an *add* operation to a Set CRDT.

```
1  apiRouter
2    .route('/set/:set_id')
3    .put(async function(req, res, next) {
4       /// ...

6        var setId = req.params.set_id;
7        var value = req.body.value;

9        let tx = await atdClient.startTransaction();
10       let set = tx.set(setId);
11       await tx.update(set.add(value));
12       await tx.commit();

14     // ...
15       res.json({
16         status: 'OK'
17         // ...
18       });
19     // ...
20   })
```

As we can see from the **Listing** 5.2, this code looks very similar to what we have already seen before, with some small differences. At the *line 7*, for example, we receive an element (the one, which has to be added to the Set CRDT identified as *id*) passed by a client and assign it to the variable *value*. Afterwards, we start a transaction and at the *line 10* assign to the variable *set* an Antidote Object associated with *setId*. Later, we call an operation *update* of the transaction object *tx*, which let us update the value in the database. We passed it as a parameter a method call *add(value)* of the object *set*. Finally, we commit the transaction and send back to the client the status of request. Additionally, even though it is not the case for the code we presented in the above scenarios, the same transaction in

AntidoteDB can mix read and update methods on different objects.

Applying the operations performed offline

Now, let us have a look at the server's logic when it comes to synchronising operations, which were performed at the client offline and were sent to the server when the internet connection was re-established.

Listing 5.3 Code for applying an *add* operation to a Set CRDT.

```
apiRouter.route('/set_sync/:set_id').post(async function(req,
    res, next) {
  // ...
  var setId = req.params.set_id;
  var lastCommitTimestamp = req.body.lastCommitTimestamp;
  var updates = req.body.updates;

  setTimestamp(lastCommitTimestamp, false);
  let tx = await atdClient.startTransaction();
  let set = tx.set(setId);

  var antidoteUpdates = [];
  updates.forEach(element => {
    if (element.type === 'add') {
      antidoteUpdates.push(set.add(element.value));
    } else if (element.type === 'remove') {
      antidoteUpdates.push(set.remove(element.value));
    }
  });

  await tx.update(antidoteUpdates);
  await tx.commit();

  // ...

  res.json({
    status: 'OK'
  });
 // ...
});
```

In **Listing 5.3**, we can see the server's implementation for this case. The most, we are interested in a part, which starts at the *line 11*. There, we create an empty array named *antidoteUpdates*, where we are going to store the updates from the client in the order in what they were received. To do that, we start iterating over the elements of the array *update* using the loop *forEach*. The array *updates* was received through the POST request from the client. The format of *updates* array allows to distinguish between the operations applied on the Set CRDT – it is either *add* or *remove*. We can see it inside the loop between the lines *13* and *17*. Afterwards, we have the array, consisting of Antidote-compatible updates. Then, the important point is to apply these updates at the timestamp, which the client had on its side, as there could be other updates coming from different clients which could have been online at that time. For this reason, at the *line 4*, we store a timestamp coming from the client, which is the latest it received from the server before going offline. Thus, we apply the updates on that timestamp, to satisfy causal consistency guarantees.

5.1.3 Client

In order for the user 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.

For each of the supported CRDTs, there are different commands available. At **Figure 5.4**, we can observe the commands that are available for a Set CRDT in our demonstration application. For the explanation of the functionality, there are three inputs available to the user. The first input labelled *Set Id*, is responsible for the names of the elements stored in the local cache and the AntidoteDB. The second input, *Element*, is there for the user to enter a list of elements to add or remove from the Set CRDT. The last input gives an opportunity to provide a specific timestamp, in order to be able to get the data at that timestamp or, in the other case, to apply a selected operation at that timestamp. Apart from the inputs, there are four buttons, which perform specific operations on a set CRDTs – *getting the value by id*, *adding and removing elements*, *removing element by id from the cache*.

However, for all this functionality to work as intended in both modes – offline and online – at first, a service worker has to be set up.

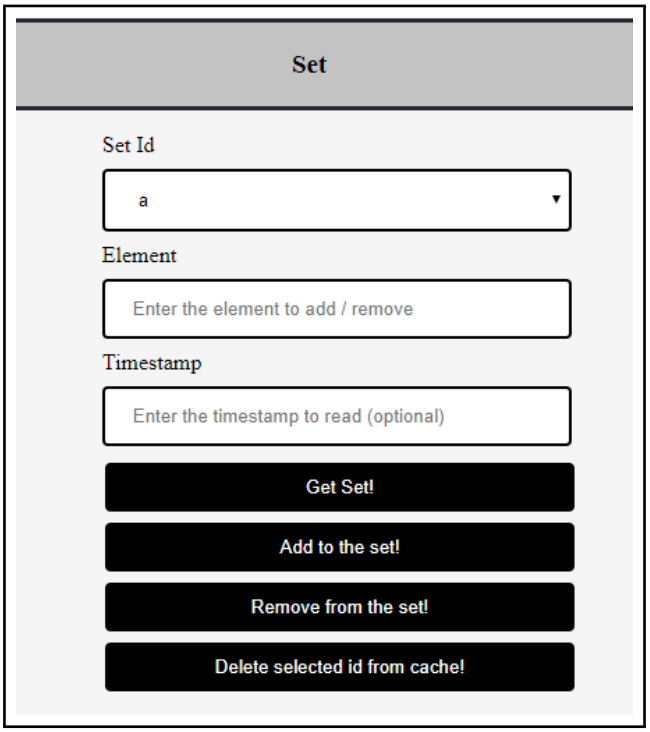


Figure 5.4: A Set CRDT part of the demonstration application, based on WebCure.

Setting up a service worker

Our service worker is located in the root directory of the application under the file *sw.js*. With its help, the application can maintain its main features such as support the offline work and synchronising the changes performed offline with the primary database. As explained in **Chapter 4**, we registered the service worker on *load* event of the application. Then, we added two listeners to the service worker for the events *install* and *fetch*, which we are going to explain further.

Listing 5.4 Code for caching necessary data for the client.

```
1 self.addEventListener('install', function(event) {
2   // Mention URLS that need to be cached
3   // It is required in order for the application to work
      offline
```

```
4   var urlsToCache = [
5     // root
6     '/',
7     '/index.html',
8     // js
9     '/logger.js',
10    '/main.js',
11    '/dbhelper.js',
12    '/idb.js',
13    // js CRDTs
14    '/CRDTs/CounterCRDT.js',
15    '/CRDTs/SetCRDT.js',
16    '/CRDTs/MVRegisterCRDT.js',
17    // css
18    '/styles.css',
19    // images
20    'img/icon-192.png',
21    'img/icon-512.png',
22    'img/favicon.ico',
23    // manifest
24    'manifest.json'
25  ];

27  event.waitUntil(
28    caches.open(CACHES_NAME).then(function(cache) {
29      // Add all mentioned urls to the cache, so the app
            could work without the internet
30      return cache.addAll(urlsToCache);
31    })
32  );
33 });
```

In **Listing 5.4**, we can see that the array *urlsToCache* consists of elements of type JavaScript strings, which are all the required frameworks, libraries, HTML pages, CSS styles and images needed for the application to work properly. This array is later used at the *line28* to create a cache storage *CACHES_NAME*, where the responses to stored URLs from the array *urlsToCache* are going to be stored. All this work is performed when the installation of the service worker for the page is triggered, which in our case is the start of the application.

Listing 5.5 Code for maintaining the requests of the application.

```
1 self.addEventListener('fetch', function(event) {
2   event.respondWith(
3     caches.match(event.request).then(function(response) {
4       // Respond the data from the cache, if it was found
            there. Otherwise, fetch from the network.
5       return response || fetch(event.request);
6     })
7   );
8 });
```

Now, once we already have the cached data, we still need to make use of it to make our application work offline. If we look at **Listing 5.5**, we can see that whenever there is a request going to the network, at *line 3* we are trying to match that request with ones we have in cache: if it is the case, then the cached object is returned or, otherwise, the fetching process from the network continues, as can be seen at the *line 5*. If the requested resource is not cached and, moreover, the client is offline, then the *fetch* request executes normally and will respond with an error, as the resource will not be loaded.

Apart from caching scripts and media files, necessary for the application to work, we will also need to set up a local database to store the data at the client side.

Setting up a local database

As explained in **Chapter 4**, in the file *js/dbhelper.js* we are setting up an IndexedDB database for the client side. There, we are going to create two object stores. The first one will consist of different CRDT data items, differentiated by their id. The other object store will keep track of the timestamp, which is associated with the latest data taken from the server. The point of having that timestamp is for the client to send it with the updates performed while offline, which will give an insight to the server, at what timestamp these updates should be applied.

Listing 5.6 Creating object stores in IndexedDB for CRDTs and timestamps.

```
// ...
var stateStore = upgradeDB.createObjectStore('crdt-states',
    {
  keyPath: 'id'
 });

var snapshotStore = upgradeDB.createObjectStore('crdt-
    timestamps',  {
  keyPath: 'id'
 });
// ...
```

We can observe the logic explained above at **Listing 5.6**, where there is an object store named *'crdt-states'* created for CRDTs and *'crdt-timestamps'* for the timestamp. A *keyPath* parameter for both of them is there in order to be able to query the data by *id*.

While it is clear why do we store the states of CRDTs, it might be not the case for why it is done for the timestamps. First of all, any updates a client makes when it loses the internet connection will be stored in the cache. However, when these updates will be sent to the server, it needs to know how to apply them. To that point, the server might have already received updates from some other clients. Therefore, to make sure that the updates we send are applied at the version of the data we were dealing with locally, a client has to provide a timestamp to the server. Secondly, we store a timestamp in cache for the persistence. While we could have turned to creating a variable for this case, it would not have allowed us to use this information after closing the application. Caching solves this problem.

Implementation of abstract Set CRDT type

Next, let us introduce our implementation of Set CRDTs abstraction for the client side, which helps to maintain the data received by the server.

Listing 5.7 A class *SetCRDT*, objects of which are going to be stored in the *'crdt-states'* object store.

```
class SetCRDT {
```

```
  constructor(id, values) {
    this.id = id;
    this.state = values ? new Set(values) : new Set();
    this.type = 'set';
    this.operations = [];
    this.sentOperations = [];
  }

  processSentOperations() {
    this.operations.forEach(operation => {
      this.sentOperations.push(operation);
    });
    this.operations = [];
  }

  materialize() {
    let values = [];

    this.sentOperations.forEach(operation => {
      if (operation.type === 'add') {
        this.state.add(operation.value);
      } else if (operation.type === 'remove') {
        this.state.delete(operation.value);
      }
    });

    this.operations.forEach(operation => {
      if (operation.type === 'add') {
        this.state.add(operation.value);
      } else if (operation.type === 'remove') {
        this.state.delete(operation.value);
      }
    });

    this.state.forEach(key => {
      values.push(key);
    });

    return values;
  }

  add(valueToAdd) {
    let operation = {
      type: 'add',
```

```
46        value: valueToAdd
47      };

49      this.operations.push(operation);
50    }

52    remove(valueToRemove) {
53      let operation = {
54        type: 'remove',
55        value: valueToRemove
56      };

58      this.operations.push(operation);
59    }
60  }
```

First, let us have a look at the constructor of a *SetCRDT* class at the **Listing 5.7**, which takes two parameters – an *id* and, optionally, an array of values. The class has the following properties:

- *id* – a string corresponding to the id of the data element stored in the server's database;
- *state* – a JavaScript Set, which reflects the state of the *SetCRDT* object and behaves like sets;
- *type* – a string reflecting the datatype and is needed for the client to distinguish between different CRDTs;
- *operations* – an array, which consists of operations performed offline at the client;
- *sentOperations* – an array, which consists of operations performed offline, but which are already sent to the server;

Secondly, there are the methods, which ease the process of working with *Set CDRT* objects at the client:

- *processSentOperations()* – shifts offline performed operations to the *sentOperations* property in order to have a distinction for the operations, which are already sent to the client.
- *materialize()* – returns the current state of the CRDT Set, taking into account the operations performed offline.

- *add(valueToAdd)* – performs an offline operation of adding an element to the Set CRDT, takes *valueToAdd* as a parameter.

- *remove(valueToRemove)* – performs an offline operation of removing an element from the Set CRDT, takes *valueToRemove* as a parameter.

Listing 5.8 An example of a *SetCRDT* object, stored on a client side.

```
{
id: "a",
operations: [{type: "add", value: "c"}],
sentOperations: [],
state: Set(2) {"b", "d"},
type: "set"
}
```

Having now understood the structure of the data stored on a client side, as well as the way it is managed, let us look at the **Listing 5.8**, where an example of a Set CRDT element with an id *a* is shown. As can be seen, it has a state received from the server of *{"b", "d"}*, while also having an offline operation *add(c)* performed on it, which is still about to be sent to the server, as an array *sentOperations* is empty at this example.

As we have already explained the server's side logic earlier, in the following sections, we will only touch the topic of a client's offline work and the logic happening at the client on a transition from offline to online modes. We believe that the part, which is related to the online work of the client is already apparent to the reader, as it comes down to the simple client-server communication through predefined requests and this part was already covered when explaining the server's side.

Read

One of the crucial aspects of the client working offline as intended is storing the CRDT states received from the server. Let us further explain the logic behind the implementation of it.

Listing 5.9 Storing CRDT states in the local database after a successful request from the server.

```
1  DBHelper.crdtDBPromise
2    .then(function(db) {
3    // ...

5      var tx = db.transaction('crdt-states', 'readwrite');
6      var store = tx.objectStore('crdt-states');

8      var item = new SetCRDT(id, value);

10     store.put(item);

12 // ...

14     return tx.complete;
15   })
16   .then(function() {
17     DBHelper.crdtDBPromise.then(function(db) {
18 // ...

20       var tx = db.transaction('crdt-timestamps', 'readwrite');
21       var store = tx.objectStore('crdt-timestamps');

23       store.put({ id: 0, data: lastCommitTimestamp });

25       return tx.complete;
26     });
27   });
```

At **Listing 5.9**, *DBHelper.crdtDBPromise* at the *line 1* gives us an access to the IndexedDB database consisting the object stores we created. There, we start a transaction on a *'crdt-states'* object store and at the *line 8* we create an element of *SetCRDT* class introduced above, passing it the *id* of the Set CRDT and its *value* that was just received from the server. Then we assign it to the variable *item* and add this *item* to the object store of CRDT states using the method *put* of *store* object. Next, we close the transaction using the method *complete* of the *tx* transaction object.

Next, when we successfully stored the received state, there is another piece of information that has to be saved on the client as well. This time it

is a timestamp associated with the update we received. Looking again at the **Listing 5.9**, at *line 20* we refer to the *'crdt-timestamps'* object store this time. Similarly as before, as can be seen at the *line 25*, we store *lastCommit-Timestamp*, which consists the timestamp received from the server. That happens each time we get a new update from the server and normally, there is no way to observe a new update without an updated timestamp as well.

Listing 5.10 Reading CRDT states from client's cache.

```
1 DBHelper.crdtDBPromise.then(function(db) {
2 // ...

4   var index = db.transaction('crdt-states').objectStore('crdt
        -states');

6   return index.get(id).then(function(state) {
7     if (state) {
8       Object.setPrototypeOf(state, SetCRDT.prototype);
9       log('[Offline] The value of ' + name.value + 'is: [ ' +
            state.materialize() + ']');
10    } else {
11      log('[Offline] Selected key is not available offline.');
12    }
13  });
14 });
```

Now comes the part regarding reading the states of the Set CRDTs from the cache. We can have a look at the **Listing 5.10**, where the code already looks familiar to us. There, at the *line 6*, we use the method *get* of *index* to search by the property *id* of the object. Then, if an element with such *id* was found in the object store, we are going to have it in a *state* variable. As we do not store the class information in the client's database, we will have to "remind" the object we have in the *state* variable about its prototype. That is why a method *Object.setPrototypeOf* is used with *state* and *SetCRDT.prototype* as parameters. After that, the object *state* will have an access to *SetCRDT*'s methods. For the demonstration of the client's functionality, in this thesis we are using a JavaScript Logger[16] library, which

[16]http://www.songho.ca/misc/logger/logger.html

we have an access to under the *log* variable at *lines 9 and 11*. It let us keep the track of neccessary information in a convinient manner. As can be seen, at the *line 9* we log the actual state of the Set CRDT using a *materialize()* method of *SetCRDT* class.

Add / Remove

Another functionality that a client covers, apart from reading and storing the values in the local database, it is performing the operations on CRDTs offline.

Listing 5.11 Performing an operation *add* on a Set CRDT while the client is offline.

```
1 DBHelper.crdtDBPromise
2   .then(function(db) {
3 // ...

5     var index = db.transaction('crdt-states').objectStore('
          crdt-states');

7     return index.get(id).then(function(storedValue) {
8       var tx = db.transaction('crdt-states', 'readwrite');
9       var store = tx.objectStore('crdt-states');

11      Object.setPrototypeOf(storedValue, SetCRDT.prototype);
12      storedValue.add(value);
13      store.put(storedValue);

15      return tx.complete;
16    });
17  });
```

Looking at **Listing 5.11**, we can see that there is not so much of a difference with previous code that we have seen on performing the read operations from IndexedDB. Again, as we can see from the example, a transaction on a *'crdt-states'* object store is created and if an element with *id* is found in the object store, its value will be available under the variable *storedValue*. Then, similarly, the method *Object.setPrototypeOf* is used for the *storedValue*, in order for it to have an access to the *SetCRDT* class me-

thods. Once it is done, at *line 12* we use the method *storedValue.add(value)*, where *value* variable is the value entered by the user. If we get back to **Listing 5.7**, we will recall that the method *add* will add an element to the array-type property *operations* of the object *storedValue*. Finally, we use *store.put(storedValue)* to put the update data item back to the client's database and afterwards complete the transaction. Similarly, the same happens when a user tries to remove elements from Set CRDTs.

A transition from offline to online

Next, we would like to discuss what happens when the client switches from offline mode back to online. For every offline operation performed on sets, we register a unique tag named *syncSetChanges*, as was explained in the **Section 4.5**.

Listing 5.12 A function *pushSetChangesToServer* triggered every time when a client re-connects to the network.

```
1  function pushSetChangesToServer() {
2    // ...
3    DBHelper.crdtDBPromise.then(function(db) {
4      var index = db.transaction('crdt-states').objectStore('
           crdt-states');

6      return index
7        .getAll()
8        .then(function(objects) {
9          DBHelper.crdtDBPromise.then(function(db) {
10           // ...
11           var index = db.transaction('crdt-timestamps')
                 .objectStore('crdt-timestamps');
12           return index.get(0).then(function(timestamp) {
13             if (objects) {
14               objects.forEach(object => {
15                 if (object.operations.length > 0) {
16                   fetch(`${DBHelper.SERVER_URL}/api/set_sync/
                         ${object.id}`, {
17                     method: 'POST',
18                     body: JSON.stringify({
19                       lastCommitTimestamp: timestamp ?
                             timestamp : undefined,
```

```
20                          updates: object.operations
21                        }),
22                        headers: {
23                          'Content-Type': 'application/json;
                                charset=utf-8'
24                        }
25                      });
26                    }
27                  });
28                }
29              });
30            });
31          })
32          .then(function() {
33            return DBHelper.crdtDBPromise.then(function(db) {
34              // ...

36           var index = db.transaction('crdt-states').objectStore(
                'crdt-states');

38           return index.getAll().then(function(objects) {
39             var tx = db.transaction('crdt-states', 'readwrite');
40             var store = tx.objectStore('crdt-states');
41             if (objects) {
42                  objects.forEach(object => {
43                    if (object.type === 'set') {
44                      Object.setPrototypeOf(object,
                            SetCRDT.prototype);
45                      object.processSentOperations();
46                      store.put(object);
47                    }
48                  });
49                }
50             return tx.complete;
51              });
52            });
53          });
54    });
55  }
```

Then, a *sync* event is going to be triggered inside the service worker. At the time it happens, we are going to check whether our tag *syncSetChanges* registered it. In this case, a function *pushSetChangesToServer* is called,

which we can observe at **Listing 5.12**.

Let us have a closer look into it. At the *line 7*, we get all the elements stored in the object store of *'crdt-states'*. In this section, we are describing the implementation of maintaining only Set CRDTs. However, the object store *'crdt-states'* normally can contain any CRDTs. Once we got the states, at *line 12*, we are also getting the latest timestamp that was stored on a client side (we use *get(0) here, as we store every new timestamp under the id 0*). This step is specifically needed for the causality of updates (more details are available in the **Chapter 3**). Then, for every Set CRDT, we are creating a POST HTTP request, which contains the information about the timestamp at which these updates were performed, as well as operations themselves. We can see it at *lines 16–24*, where the timestamp is sent as *lastCommitTimestamp* and operations are sent as *object.operations*. In such a way, this information is sent to the server, which continues processing it on its side.

However, this is not it for the client yet. There is a reason why the client has to distinguish between performed offline and already sent to the server operations. First of all, in order to not send the same updates multiple times to the server. At the *line 46* of the **Listing 5.12**, we see that the method *processSentOperations()* is called. We can look up its implementation again at the **Listing 5.7**. As we remember, it shifts already sent operations to another array named *sentOperations*, which is also a property of the class *SetCRDT*. Our reader might also think about the possibility of just clearing these operations from the cache as soon as the internet connection at the client side is re-established again. That that would not be a good idea. Let us imagine the following situation: a client works offline for some time, then the connection gets re-established, but before the client could request new updates from the server, the connection gets off again. What happens in such a scenario? First of all, the operations performed offline at the client side will be sent to the server immediately, but if we delete sent operations from the cache, then we are risking losing the data at client side. As we did not yet receive the latest updates from the server, the client would not be able to continue working offline on its data. It was the second reason, why it is better to keep offline operations at client side in two separate collections, at least before the server sends back data updates, having already applied the operations the client sent. At that point, it will be safe to remove them.

Finally, we have covered so far every aspect of the implementation, and in the next chapter, we will do an evaluation of our work.

6 Evaluation

In this chapter, we are going to evaluate the system we built, as well as present a running application based on WebCure.

6.1 Testing

To assure the correctness of our application, we covered it with test cases of different types.

Firstly, we did unit testing for the implemented CRDT classes, which purpose is to validate the correctness of their work according to the design specifications.

Listing 6.1 Simple unit test that checks the correct initialization of objects of a *SetCRDT* class.

```
1  it('Check the initialization of a SetCRDT Class', function(
       ) {
2    var a = new SetCRDT('a', ['a', 'b', 'c']);
3    var b = new SetCRDT('b');

5    expect(a.id).toEqual('a');
6    expect(a.state).toEqual(new Set(['a', 'b', 'c']));
7    expect(a.type).toEqual('set');

9    expect(b.state).toEqual(new Set([]));

11   expect(a.operations).toEqual([]);
12   expect(a.sentOperations).toEqual([]);
13 });
```

At the **Listing 6.1**, we can observe a simple unit test, which helps to check the correctness of the initialization of *SetCRDT* objects, written with

a help of Jasmine framework[17].At *lines 9 and 10* we can see that objects *a* and *b* are created using a constructor of *SetCRDT* with parameters. The first parameter, as our reader might remember from **Section 5.1.3**, is referring to the *id* of the set, while the second one is referring to the elements it contains initially. Later, step by step, every property is checked according to the expected values it should possess. This is the way we wrote unit tests for the abstract CRDT classes we implemented for the client side.

Apart from that, we did system testing as well. To achieve that, we had to mimic the behaviour of WebCure in the testing environment. For that, we ...

- add the comparison between your approach and the other, in related work.

[17]https://jasmine.github.io/

7 Related Work

In this chapter, we present some other approaches, which influenced our work.

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

Geo-replication of data into several data centres (DC) across the world is used in cloud platforms in order to improve availability and latency[**?**]. 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 concerning the increasing availability of systems.

A system 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[**?**]. 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, SwiftCloud distributed object database is the first to provide fast reads and writes via a causally-consistent client-side local cache backed by the cloud. To provide the convergence of the data, SwiftCloud relies on CRDTs, which have rich confluent semantics[**?**].

7.2 Legion: a framework, which enriches web applications

A framework named Legion shows another exciting approach on how to address availability and scalability issues by using the cache at the client-side. The idea is to avoid a concept of a centralised infrastructure for mediating user interactions, as it causes unnecessarily high latency and hin-

ders fault-tolerance and scalability[**?**]. As an alternative, authors of Legion suggest client web applications to securely replicate data from servers and afterwards synchronise 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**.

7.3 Developing web and mobile applications with offline usage experience

Nowadays, applications with offline experience are becoming extremely popular. In this thesis, in **Chapter 4** we presented an approach of developing web applications with the help of service workers and background synchronisation (they also go by the 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 specific frameworks are used. In the following subsections, we are going to describe the 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 creating custom HTML tags to use in web pages[**?**]. 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[**?**], which repeats the solution used in this thesis. Nevertheless, in contrast to the implementation offered in this thesis, working with Polymer App Toolbox requires additional knowledge of the Polymer framework. Currently, among the users of Polymer, apart from Google, there are such giants as Electronic Arts, IBM and Coca-Cola[**?**].

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 a fast load of resources from the hard drive[**?**]. However, from 2015 the application cache is considered to be deprecated and is recommended to be avoided[**?**] in favour of Service Workers. Thus, for the implementation of WebCure, we favoured the approach of using a combination of a service worker and a Cache API.

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 will only stop on it providing offline experience for applications that are developed using Hoodie. The documentation states that the framework is offline-first[**?**], which means that all the data is firstly stored locally and could be accessed without the internet connection. It is possible due to PouchDB database, which performs this work in the background[**?**]. 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 regarding storing data on the client-side. It uses IndexedDB, localStorage or WebSQL with a simple API. localForage sits on top of the data store 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[**?**]. localForage might be a useful library; however, as we already used IndexedDB Promised to work with IndexedDB database, we did not need to use anything heavier, such as localForage, primarily as it does not provide any fundamental differences in the approach.

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 synchronisation with CouchDB-compatible servers. CouchDB, in its turn, is a database that is supported by a replication approach, which allows synchronising two or more servers, based on CouchDB. As there is a replication approach, it has its way of dealing with conflicts, which we explain next by following the description of the protocol offered by **?**].

Listing 7.1 A typical result of retrieving the item *document* stored in CouchDB.

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

For every item stored in CouchDB, the database will add two extra properties: *_id* and *_rev*, which can be seen at **Listing 7.1** that shows the result of getting an element named *document* previously stored in the database. As we 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.

Listing 7.2 Updating the value of item *document* by adding *b* into it.

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

For example, to change the value of *document* by adding *b* into it, a simple PUT HTTP-request should be sent, as **Listing 7.2** shows.

Listing 7.3 The result of requesting the updated version of *document*

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

The next retrieval of the *document* will get us the result shown at **Listing 7.3**.

As we 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.

Listing 7.4 Updating the value of item *document* by adding *d* into it.

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

However, as our reader might have guessed, it is much more interesting, when we have more than one server. Let us assume that now we have two CouchDB servers. Imagine we try to update the *document* once again with the values shown at **Listing 7.4**.

Listing 7.5 The result of requesting the *document* from CouchDB-1.

```
1 GET /database/document
2 {"_id":"document","_rev":"3-2235fd4815b81b2da1b84159aba4006e"
    , "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 7.5**.

Listing 7.6 The result of requesting the *document* from CouchDB-2.

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

However, the second CouchDB server still has the old data, as we can see at **Listing 7.6**.

Ideally, the replication happens fastly, and both databases will synchronise and reach the same state. However, sometimes it might take a while. Moreover, 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 7.7 Updating the value of item *document* by adding element *e* and removing previously added element *d*

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

Listing 7.8 The demonstration of a conflict situation happening, when the *_rev* of sent operation and the one at the server do not match.

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

As we mentioned earlier, the response of the second CouchDB server for the operation shown at **Listing 7.7** will look like the one at **Listing 7.8**.

Listing 7.9 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.

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

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 7.9**.

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;

Let us now summarize the advantages WebCure has due to using AntidoteDB and not CouchDB.

First of all, since CouchDB "blocks" the possibility of updating the server's database without knowing the latest revision *_rev*, it already adds extra-work in the design and implementation of the protocol. Moreover, as we believe, it does not allow to perform concurrent updates. AntidoteDB, on the other hand, uses the concept of timestamps, which allows supporting causal consistency guarantees and, as it is not creating such restrictions on making updates, it has a massive advantage with the support of concurrent updates.

Realm Mobile

Realm Mobile is a framework, which makes integration of a client-side database for iOS and Android with a server-side, which offers the following features: real-time synchronisation, 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;
- **The 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. It means that if both sides append items to the end of a list, they will end up in order of insertion time;”[**?**]

The authors of the framework state that “strong eventual consistency” guarantees are reached[**?**] with the approach mentioned above. 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.

8 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.)

8.1 Summary

8.2 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

Listings