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

Technische Universität Kaiserslautern

Zusammenfassung

Zusammenfassung auf deutsch

Abstract

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

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

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

Acknowledgement

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

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

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

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

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

Kaiserslautern, den 15. December 2018 Server Khalilov

Contents

1 Introduction

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

1.1 Motivation

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

1.2 Research questions

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

- **RQ1.** How efficient is it to use a web-client with cache rather than without it?

- **RQ2.** What are the methods available to implement web-applications that would be able to work off-line and in the conditions of poor network connections?

- **RQ3.** What could be a scalable solution for transmitting CRDT data between a server and clients?

1.3 The structure

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

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

2 Theoretical background

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

2.1 Main concepts

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

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

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

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

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

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

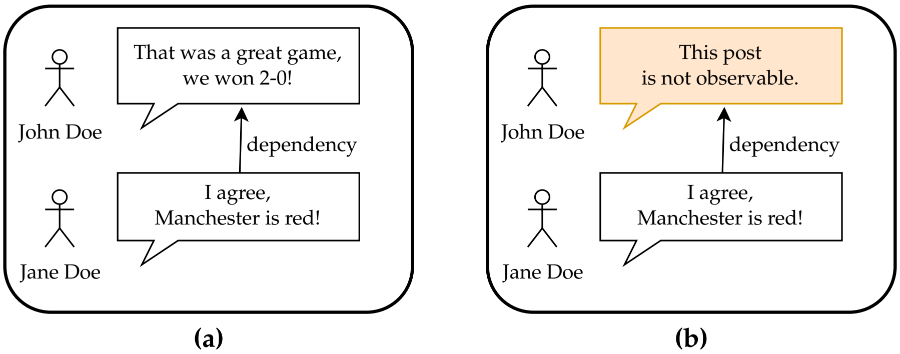


Figure 2.1: An example of how a causally consistent behaviour (a) and the one that is not (b) could work in a social network.

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

2.2 AntidoteDB

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

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

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

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

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

2.3 Conflict-Free Replicated Datatypes (CRDTs)

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

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

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

Operation-based replication approach

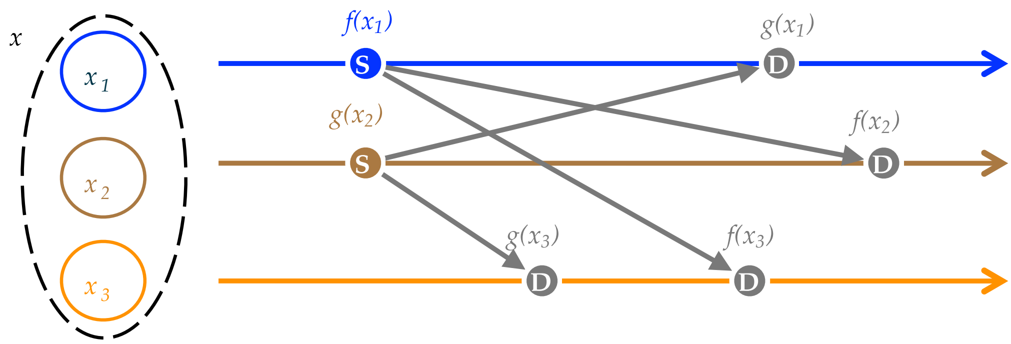


Figure 2.2: Operation-based approach[12]. «S» stands for source replicas and «D» for downstream replicas.

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

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

State-based replication approach

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

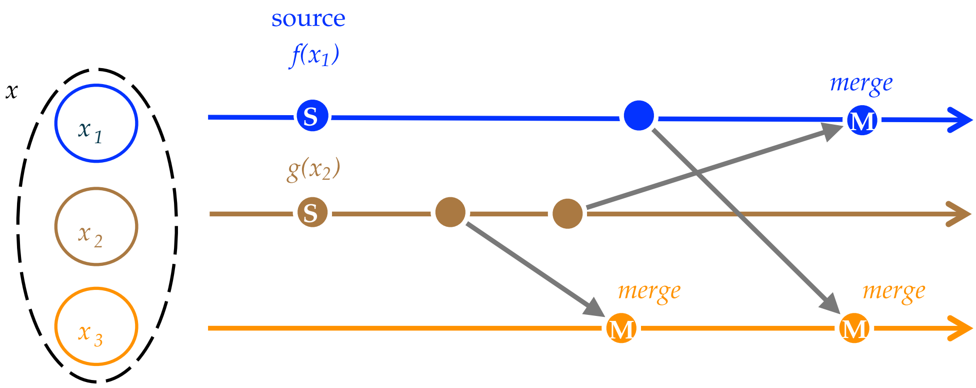


Figure 2.3: State-based approach[12]. «S» stands for source replicas and «M» for merging stages.

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

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[13]. 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 concur-

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

rent 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}*, as *add* operation wins. A *remove* operation will "overwrite" an *add* operation only when it happens after it[13].

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[14].

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

3.2 Assumptions

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

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 the 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.

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	Do not store the element's values in the cache, if they were requested at specific timestamp (for the matter of having only the latest data on the client's side).
R6	The user should be able to remove from the client any stored data element.
R7	It should not be possible to create elements of different CRDTs with the same name (due to limitations of AntidoteDB).
R8	When offline, it should be possible to perform operations on supported CRDTs.
R9	Any operations performed offline, once the connection is restored, should be sent to the server immediately.
R10	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 web application should be available online and offline (except for the functionality with timestamp-related updates).
NFR2	The web application should be available with a poor internet connection.

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 the 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. Due to the time constraints and the amount of work, this thesis will consider only the operation-based approach. 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

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

[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.[15]

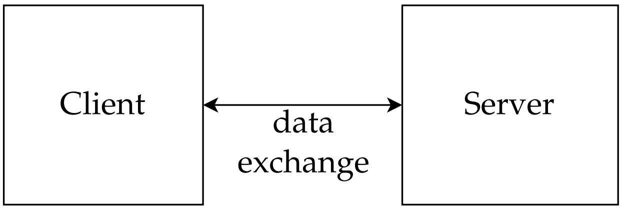


Figure 3.1: An overview of the communication protocol.

stages of their communication and describe, how we handled these processes.

Graphics notations

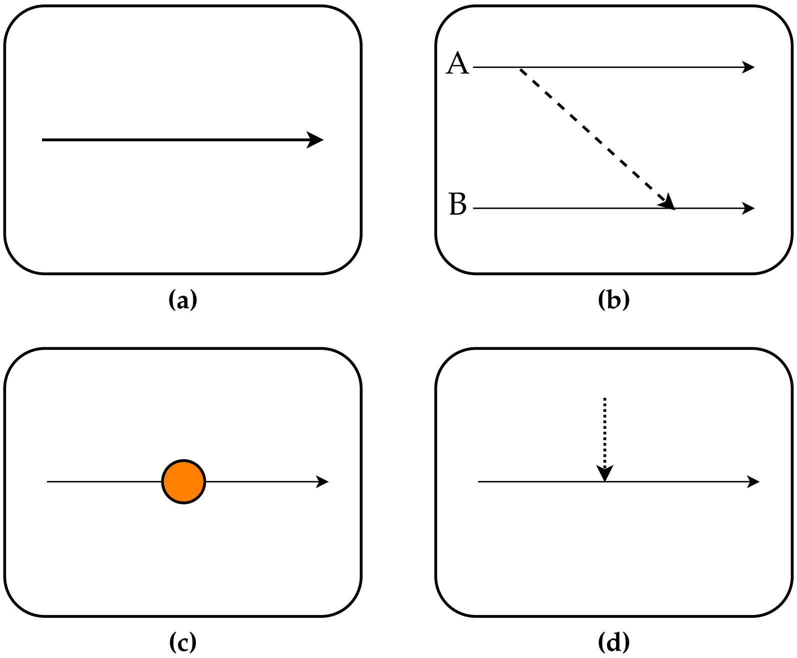


Figure 3.2: An overview of notations used in the following chapters for the protocol explanation.

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, every point of which represents a different timestamp. 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 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

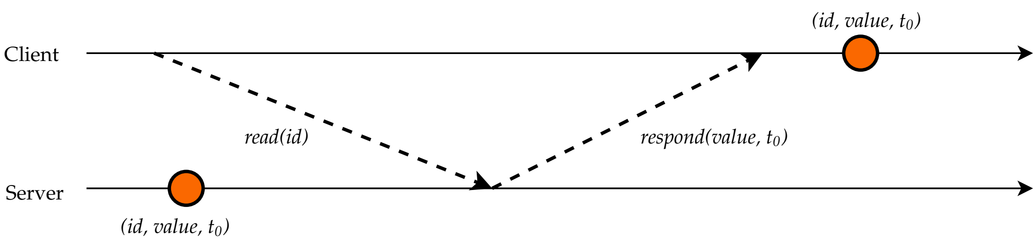


Figure 3.3: The communication between a client and a server for the read function.

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) {
  if (ONLINE) {
    value = fetch(id); // request the value of the data by id
        from the server
    store(value, timestamp); // store the value and timestamp
        received from the server on a client's storage
```

```
    print(value, timestamp); // print the value and timestamp
        received from the server to the user
  } else {
    value = fetch(id); // Get the value of the data by id
       from the cache
    print(value); // print the value to the user
  }
}
```

The pseudocode of the logic for this *read* functionality can be seen at **Listing 3.1**. If a client is online, then the value and timestamp associated with passed *id* are going to be fetched from the server and, after that, stored on a client's storage and displayed to the user. However, if the client is offline, then the client will try to get the data from the cache for the requested id, or, if there is none, it will show the appropriate message to the user.

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.

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

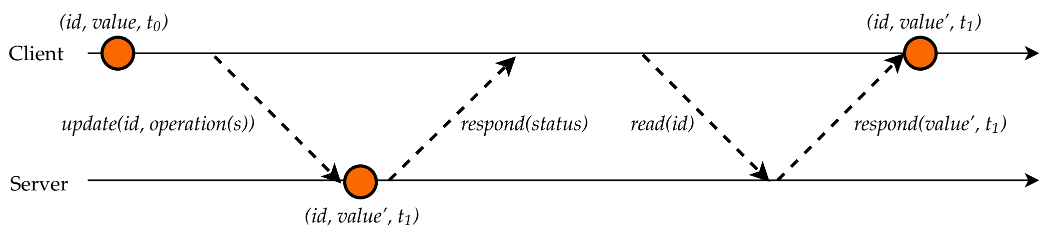


Figure 3.4: The communication between a client and a server for the update function.

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.

```
// Update function that processes user-made update
// @param id: an id for the object that should be updated;
// @param op: operation performed on the object for the specified id
function update(id, op){
    send(id, op); // send an operation to the server for the element id
    if (SUCCESS){
        print("The operation was applied on the server successfully");
    }
    else {
        store(op); // store the operation on a client's side in order to try sending it again later
    }
}
```

However, let us look at the pseudocode placed at **Listing** 3.2. There we can see the function *update*, which has to have access to the parameters *id* and *op* that might be taken from the presentation layer of the system. Afterwards, an attempt is happening to send the operation *op* for the element *id* to the server. If it succeeds, then a success message will be printed. Otherwise, the operation will be stored locally on a client's side. That makes the update available while the user is offline and gives an opportunity to send the operation again when the connection is back again.

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

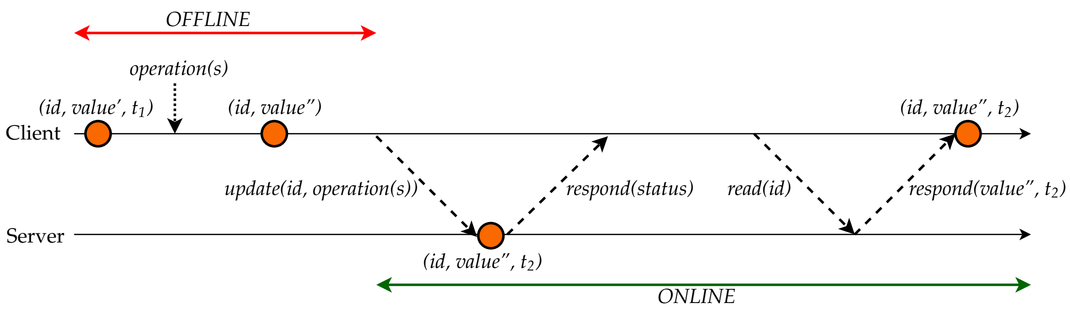


Figure 3.5: The communication between a client and a server while offline with a transition to online.

Offline behaviour

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.

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

```
        });
    }
}
```

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.

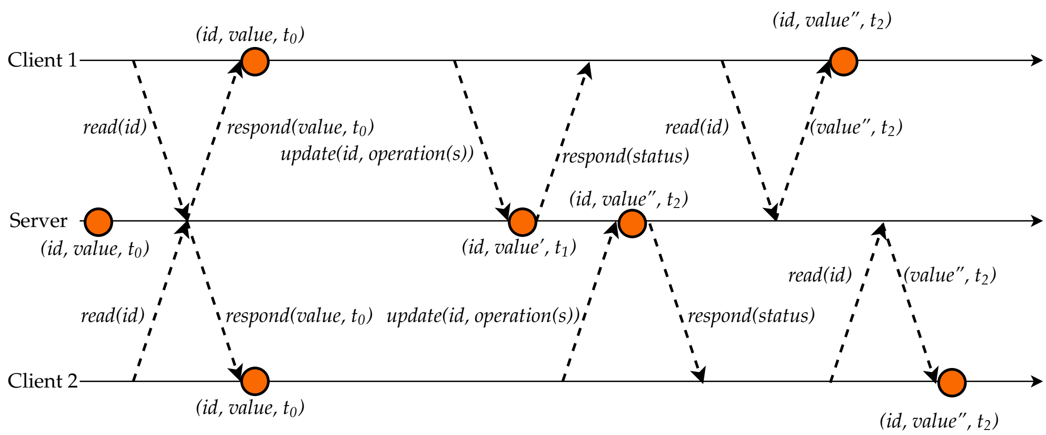


Figure 3.6: The communication between two clients and a server.

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[16] 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[17] 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[19], 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[18].

[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;

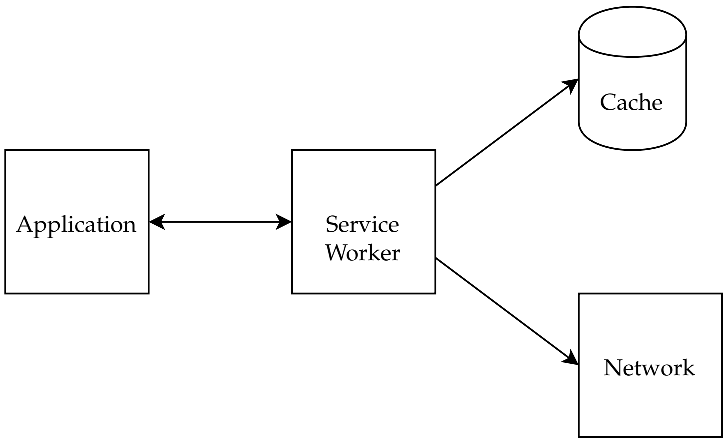


Figure 4.1: An overview of the service worker being able to provide the application experience of both online and offline modes.

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[20].

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[19].

```
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[21].

and Firefox support it. However, it is still only partially supported in Opera and Safari, and not at all in the Internet Explorer[22] – 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[19].

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

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[19].

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

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[19].

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

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*[23].

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

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[25]. 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[24].

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[25].

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

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[26, 27]. 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][28]. 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[29, 30], 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*[31] 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[31].

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

// Then later, request a one-off sync:
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.

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

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[31].

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

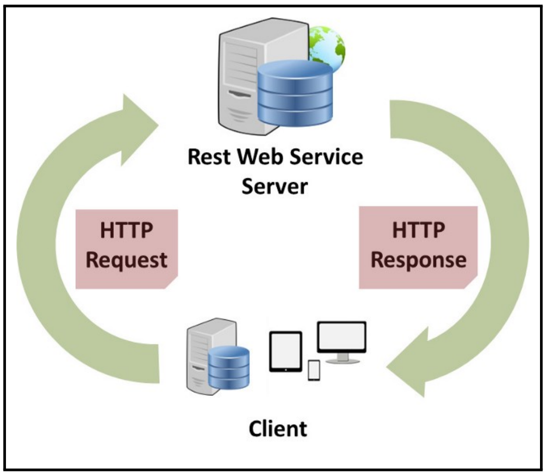
Afterwards, when a page controlled by the service worker is going back online (which is when the user agent has established a network connection[32]), a *sync* event is triggered. There, it is possible to perform a distinct action, depending on the registered earlier tag. For instance, at **Listing 4.8** 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[31].

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 significant obstacles that we faced during the development phase.

Addressing the client, in the **Chapter 4** we already named the key frameworks and techniques that we used in the implementation. Concerning the server, which is going to provide to the client the RESTful[9] interface, we will cover the use cases.



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

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

To demonstrate the functionality of the WebCure, we covered three different types of CRDTs used in AntidoteDB: a Counter, a Set and a Multi-value Register. We managed to 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[33].

5.1 System's main components

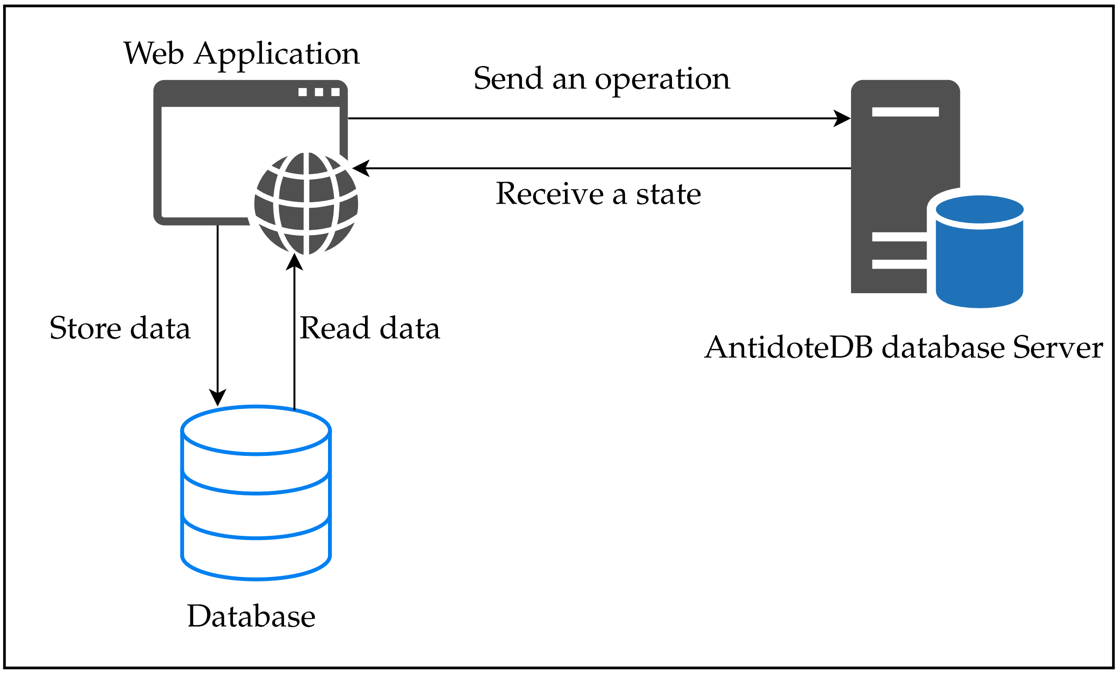


Figure 5.2: A top-level view of the system's design.

In the chapter **Chapter 3**, we introduced a client and a server as black boxes, without really knowing what is happening inside each of them. However, from now on, we have a clear understanding 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, possesses a setup with AntidoteDB and, apart from that, can exchange messages with a client. Regarding the exchanging of the data, as was mentioned before, 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 implement the WebCure using a 3-tier architecture[34], as it suits the design we introduced. Its principal distinction is that it has a clear separation between the presentation, application and data layers, as we can spot at the **Figure 5.3**, where we also noted how each JavaScript file of the WebCure relates to these layers. Each layer is responsible for their tasks. Let us now briefly characterise each of them. A *presentation layer* is

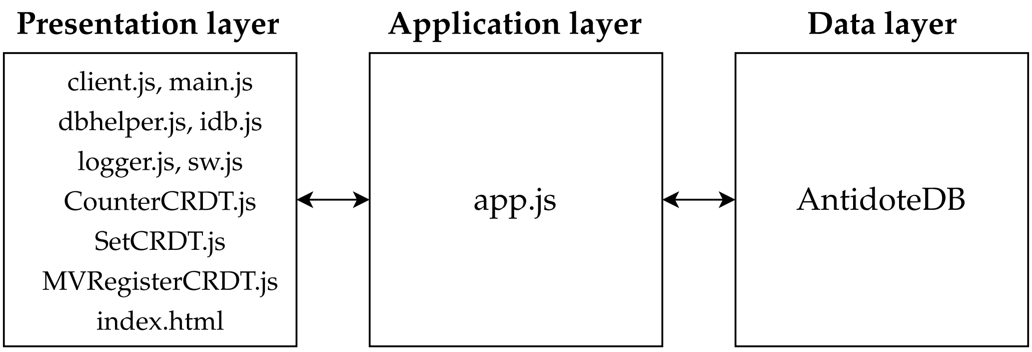


Figure 5.3: 3-Tier architecture.

responsible for presenting the information to the users, while it can also accept requests from them. An *application layer*, in its turn, implements the algorithmic part of the system and answers the operations requested by the clients. 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[35].

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

5.1.1 Database layer

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[36], which suits the stack of the technologies we are using for WebCure. The database itself will be running on a Docker[10] container created from an image of 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 be able to read and to apply changes to the data at a specific timestamp, the Antidote image for docker should be running with certifications checks disabled, while the Antidote JavaScript client was added a flag to make sure that the client supports the possibility of making transactions at specific timestamps. Apart from that, the database was also changed to support this functionality. The data in AntidoteDB is stored as a binary data and, therefore, the Antidote JavaScript Client performs the serialisation process.

[10] https://www.docker.com/

5.1.2 Server

The server side of the 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[11] framework called Express[12]. As we are taking Set CRDT as an example, let us mention the methods the server has for it:

- 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 the Set CRDT;
- Applying all the add and remove 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 back the data requested by the client.

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

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

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

    setTimestamp(timestamp,..);

    let tx = await atdClient.startTransaction();
    let set = tx.set(setId);
    let val = await set.read();

    await tx.commit();
```

[11]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/.

[12]Express is a Nodej.s framework that provides features for web and mobile applications. More details can be found here: https://expressjs.com/.

```
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 5*, we can observe the logic of the function. There, firstly, we assign variables the passed parameters. The id of the requested Set CRDT is assigned to the variable *setId* and an optional timestamp, if it was passed, is assigned to the variable *timetamp*, which is used in the function *setTimestamp* to set the current timestamp of the database. That let us making updates at any timestamp and maintain the sequence of updates.

At *line 10* 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[13], for a successful transmission in a JSON format, it is converted to the Base64[14] at *line 20*) is sent back to the client.

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.

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

[14] 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[37].

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 now get the passed by a client value and assign it to the *value* variable. Afterwards, we start a transaction and at the *line 10* assign to the variable *set* an Antidote Object associated with *setId*. Later, we perform an operation *update* of the transaction object *tx* and call the method *add* of object *set*, passing it the *value*. Finally, we commit the transaction and send back to the client the status of request.

Applying the operations performed offline

Now, let us have a look at the server's logic when it comes to synchronising the 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.

```
1 apiRouter.route('/set_sync/:set_id').post(async function(req,
      res, next) {
2     // ...
3     var setId = req.params.set_id;
4     var lastCommitTimestamp = req.body.lastCommitTimestamp;
5     var updates = req.body.updates;

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

11    var antidoteUpdates = [];
12    updates.forEach(element => {
13      if (element.type === 'add') {
14        antidoteUpdates.push(set.add(element.value));
15      } else if (element.type === 'remove') {
16        antidoteUpdates.push(set.remove(element.value));
17      }
18    });

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

23    // ...

25    res.json({
26      status: 'OK'
27    });
28   // ...
29 });
```

At the **Listing 5.3**, we can see the server's implementation for this case. We are interested the most in part, which starts at *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 using *forEach* over the elements of the array *update*, which was received through the POST request from the client. The updates from the client arrive at the server's side in such a format, which allows distinguishing between the operations applied on the Set CRDT – either *add* or *remove*. We can see it inside the loop between the lines *13* and *17*. Afterwards, we have the array, consisting 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 be able to communicate with an AntidoteDB server, we are going to have a running web application that serves as a client. It runs in the web-browser, supports various commands from the user and sits on top of the local database layer.

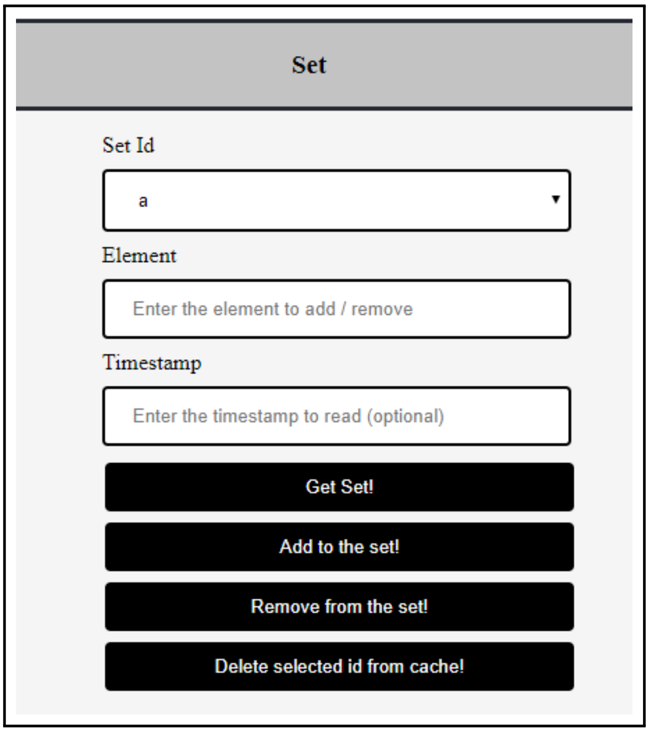


Figure 5.4: A part of the WebCure interface on working with Sets.

For each of supported CRDTs, there are different available commands, which we can observe from the **Figure 5.4**. For the demonstration 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.

Setting up a service worker

Our service worker lies in the root directory of the application under the file *sw.js*. Due to it, the application can maintain its main features such as support the offline work and synchronising the changes performed offline with the primary database. As it was explained in the Technologies chapter, 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.

```
self.addEventListener('install', function(event) {
  // Mention URLS that need to be cached
  // It is required in order for the application to work
     offline
  var urlsToCache = [
    // root
    '/',
    '/index.html',
    // js
    '/logger.js',
    '/main.js',
    '/dbhelper.js',
    '/idb.js',
    // js CRDTs
    '/CRDTs/CounterCRDT.js',
    '/CRDTs/SetCRDT.js',
    '/CRDTs/MVRegisterCRDT.js',
```

```
    // css
    '/styles.css',
    // images
    'img/icon-192.png',
    'img/icon-512.png',
    'img/favicon.ico',
    // manifest
    'manifest.json'
  ];

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

At the **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 to create a cache storage *CACHES_NAME*, where the responses to the 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.

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

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*.

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

Setting up a local database

As it was explained in the technologies chapter, in the file *js/dbhelper.js* we are setting up an IndexedDB database for the client's side. There, we are going to create two object stores. First one will consist of different CRDT data items, differentiated by 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 which 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*.

Implementation of abstract Set CRDT type

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

Listing 5.7 A class Set CRDT, 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() {
    if (this.operations.length > 0) {
      this.operations.forEach(operation => {
        this.sentOperations.push(operation);
      });
      this.operations = [];
    }
  }

  calculateState() {
    let values = [];

    if (this.sentOperations.length > 0) {
      this.sentOperations.forEach(operation => {
        if (operation.type === 'add') {
          this.state.add(operation.value);
        } else if (operation.type === 'remove') {
          this.state.delete(operation.value);
        }
      });
    }

    if (this.operations.length > 0) {
```

```
      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',
      value: valueToAdd
    };

    this.operations.push(operation);
  }

  remove(valueToRemove) {
    let operation = {
      type: 'remove',
      value: valueToRemove
    };

    this.operations.push(operation);
  }
}
```

First, let us have a look at the constructor of a Set CRDT 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 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 CDRTs 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.
- *calculateState* – returns the current state of the CRDT Set, taking into account the operations performed offline.
- *add* – performs an offline operation of adding an element to the Set CRDT, takes an element to add as a parameter.
- *remove* – performs an offline operation of removing an element from the Set CRDT, takes an element to remove as a parameter.

Listing 5.8 An example of a Set CRDT stored on a client's 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's 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 *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 server-client 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.

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

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

    var item = new SetCRDT(id, value);

    store.put(item);

// ...

    return tx.complete;
  })
  .then(function() {
    DBHelper.crdtDBPromise.then(function(db) {
// ...

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

```
23      if (temp) {
24        store.put({ id: 0, data: lastCommitTimestamp });
25      }

27      return tx.complete;
28    });
29  });
```

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 create 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 *complete* of transaction *tx* 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 *lastCommitTimestamp*, which consists the timestamp received from the server. That happens each time we get a new update from the server.

Listing 5.10 Reading CRDT states from client's cache.

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

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

  return index.get(id).then(function(state) {
    if (state) {
     Object.setPrototypeOf(state, SetCRDT.prototype);
     log('[Offline] The value of ' + name.value + 'is: [ ' +
         state.calculateState() + ']');
    } else {
     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 **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 don't 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[15] library, which 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 *line 9* we log the actual state of the Set CRDT using a *calculateState()* 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 Reading CRDT states from client's database.

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

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

    return index.get(id).then(function(storedValue) {
      var tx = db.transaction('crdt-states', 'readwrite');
```

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

```
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 the **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 the 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 methods. Once it is done, at *line 12* we use the method *storedValue.add(value)*, where *value* is the value entered by the user. If we get back to the **Listing 5.7**, we will remember that it will add an element to the array-type property *operations* of the object *storedValue*. Finally, we uset *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

One more case, the implementation of what we would like to cover is what happens when the client switches from offline mode back to online. The implementation of this part was already slightly touched in the Technologies chapter in the subsection about Background Synchronization. For every offline operation performed on sets, we register a unique tag named *syncSetChanges*.

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

```
function pushSetChangesToServer() {
  // ...
```

```
DBHelper.crdtDBPromise.then(function(db) {
  var index = db.transaction('crdt-states').objectStore('
      crdt-states');

  return index
    .getAll()
    .then(function(objects) {
      DBHelper.crdtDBPromise.then(function(db) {
        // ...
        var index = db.transaction('crdt-timestamps')
            .objectStore('crdt-timestamps');
        return index.get(0).then(function(timestamp) {
          if (objects) {
            objects.forEach(object => {
              if (object.operations.length > 0) {
                fetch(`${DBHelper.SERVER_URL}/api/set_sync/
                    ${object.id}`, {
                  method: 'POST',
                  body: JSON.stringify({
                    lastCommitTimestamp: timestamp ?
                        timestamp : undefined,
                    updates: object.operations
                  }),
                  headers: {
                    'Content-Type': 'application/json;
                        charset=utf-8'
                  }
                });
              }
            });
          }
        });
      });
    })
    .then(function() {
      return DBHelper.crdtDBPromise.then(function(db) {
        // ...

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

      return index.getAll().then(function(objects) {
        var tx = db.transaction('crdt-states', 'readwrite');
        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 case it is, a function *pushSetChangesToServer* is going to be 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 you the implementation of maintaining only Set CRDTs. However, the object store *'crdt-states'* normally can contain any CRDTs. Once we got the states, at the *line 12*, we are also getting the latest timestamp that was stored on a client's side. This step is specifically needed for the causality of updates (more details are available in the Design chapter). 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 that 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 not to 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 is re-established again. That is not a good idea, in any case. 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's side will be sent to the server immediately, but if we clear sent operations from the cache without storing it in a different collection, then we are risking losing the data at client's side. As we did not yet receive the latest updates from the server, even if the server received the operations which the client sent, while being offline again, the client would not be able to continue working on its data. It was the second reason, why it is better to keep offline operations at client's side in two separate collections, at least before the server sends back new data updates, having already applied the operations a client sent. At that point, it would be safe to remove those operations.

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

Evaluation

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

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

7 Related Work

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

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

Geo-replication of data into several data centers (DC) across the world is used in cloud platforms in order to improve availability and latency[38]. This goal could be achieved even to a greater extent by storing some part of the data or even by replicating all of it at client machines. Thus, caching could be useful in terms of increasing availability of systems.

One of such systems that integrates client- and server-side storage is called SwiftCloud, where the idea is to cache a subset of the objects from the DCs, and if the appropriate objects are in cache, then responsiveness is improved and the operation without an internet connection is possible[39]. The authors of the SwiftCloud state that it improves latency and throughput if it is compared to general geo-replication techniques. That is possible due to availability during faults thanks to automatic switch of DCs when the current one does not respond. Apart from that, the system also maintains consistency guarantees [7].

7.2 Legion: a framework, which enriches web applications

A framework named Legion shows another interesting approach how to address availability and scalability issues by using cache at the client-side. The idea is to avoid a concept of a centralized infrastructure for mediating user interactions, as it causes unnecessarily high latency and hinders fault-tolerance and scalability[40]. As an alternative, authors of Legion suggest client web applications to securely replicate data from servers and afterwards synchonize the data among the clients. This change makes the system less dependent on the server and, moreover, it reduces the latency of interactions among the clients. The guarantee of convergence between all of the replicas is possible due to CRDTs, introduced in **Chapter 2**.

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

Polymer App Toolbox

Polymer is a JavaScript library from Google, which helps to build web applications with the use of Web Components. The former concept represents a set of web platform APIs, which allow to create custom HTML tags to use in web pages[41]. 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[42], which repeats the solution used in this thesis. Currently, among the users of Polymer, apart from Google, there are such giants as Electronic Arts, IBM and Coca-Cola[43].

HTML5 Specification

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

The latter makes possible the following advantages: offline browsing, flexibility and fast load of resources from the hard drive[44]. However, from 2015 the application cache is considered to be deprecated and is recommended to be avoided[45] in favour of Service Workers. Apart from that, HTML5 also lets the developer to check in which stage the user's navigator is, by simply checking the attribute *navigator.onLine*[46]. Offline and online events make changes to this attribute, which reflects the status of current connection. The advantages of HTML5 approach is the reusability of code, shorter development and testing times and universal support of mobile platforms. However, if compared to the mobile applications written in native languages, HTML5 web applications could be inferior to them.

Hoodie

Hoodie is a framework, which eases the process of developing web and iOS applications. We will not cover all the features it offers and only stop on it providing offline experience for applications that are developed using Hoodie. The documentation states that framework is offline-first[47], which means that all the data is fistly stored locally and could be accessed without the internet connection. This is possible due to PouchDB database, which performs this work in the background[48]. We will explore the process of how PouchDB works below.

localForage

localForage is a JavaScript library, which provides the possibility to improve the offline-experience of web applications in terms of storing data on the client-side. It uses IndexedDB, localStorage or WebSQL with a simple API. localForage sits on top of datastore layer and provides a range of methods to control the data. One of the useful benefits of it is that the data is not required to be explicitly converted into JSON format, as localForage does that automatically[49].

PouchDB and CouchDB

PouchDB represents an open-source JavaScript database, which is designed to build applications, which work well online and offline. To put it in a nutshell, PouchDB enables web applications to store the data locally, and, apart from that, eases the process of synchonization with CouchDB-compatibale servers. CouchDB, in its turn, is a database that is supported by a replication approach, which allows to synchonize two or more servers, based on CouchDB. As there is a replication approach, it has its own way of dealing with conflicts, which we explain next by following the description of the protocol offered by Lehnardt [50]. 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.

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

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

As you can see, the *_id* represents the name of the item, which is a custom name set by the user, while *_rev* represents the hash value, associated with the content. Afterwards, whenever we want to change the item *document*, the same *_id* and *_rev* should be used. For example, to change the value of *document* by adding *b* into it, a simple PUT HTTP-request should be sent, as **Listing** 7.2 shows.

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

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

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

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

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

As you can see, the *_rev* property got updated. In case the wrong revision is sent to update the document, the database will respond with an error.
However, as you might have guessed, it is much more interesting, when you have more than one server. Let us assume that now we have two CouchDB servers. Imageine we try to update the *document* once again with the values shown at **Listing** 7.4.

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

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

For example, let us say that it gets written to the first CouchDB server, which gives the response shown at **Listing** 7.5.

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

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

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

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

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

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

Listing 7.7 Updating the value of item *document* by adding element *e* and removing previously added element *d*

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

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

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.

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

However, there is another option to perform this update. We might have a strategy of getting the latest *_rev* first, which was *"2-c5242a69558bf0c24dda59b585d1a52b"* at the second CouchDB server, and only then applying the update. So, the operation will look as at **Listing** 7.9.

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.

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

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

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

Realm Mobile

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

- **Deletes always win.** If one side deletes an object it will always stay deleted, even if the other side has made changes to it later on.
- **Last update wins.** If two sides update the same property, the value will end up as the last updated.
- **Inserts in lists are ordered by time.** If two items are inserted at the same position, the item that was inserted first will end up before the other item. This means that if both sides append items to the end of a list they will end up in order of insertion time."[51]

The authors of the framework state that “strong eventual consistency” guarantees are reached[51] with the above mentioned approach. Though, such way of conflict handling is not as flexible as CRDTs and has to be taken into account by the programmer at the stage of the development.

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] Datatypes in antidote. https://antidotedb.gitbook.io/documentation/api/datatypes, 2018 (accessed November 26, 2018).

[15] Black box model. https://www.investopedia.com/terms/b/blackbox.asp, 2018 (accessed November 29, 2018).

[16] Caching files with service worker. https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker, 2018 (accessed November 28, 2018).

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

[18] Web workers api. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API, 2018 (accessed November 15, 2018).

[19] Introduction to service worker. https://developers.google.com/web/ilt/pwa/introduction-to-service-worker, 2018 (accessed November 18, 2018).

[20] Yvo Desmedt. *Man-in-the-Middle Attack*, pages 368–368. Springer US, Boston, MA, 2005. ISBN 978-0-387-23483-0. doi: 10.1007/0-387-23483-7_241. URL https://doi.org/10.1007/0-387-23483-7_241.

[21] The navigator object. https://www.w3schools.com/jsref/obj_navigator.asp, 2018 (accessed November 28, 2018).

[22] Service worker api. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API, 2018 (accessed November 28, 2018).

[23] Mat Scales. Using the cache api. https://developers.google.com/web/fundamentals/instant-and-offline/web-storage/cache-api, 2018 (accessed November 28, 2018).

[24] Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise, 2018 (accessed November 23, 2018).

[25] Working with indexeddb. https://developers.google.com/web/ilt/pwa/working-with-indexeddb, 2018 (accessed November 23, 2018).

[26] Jake Archibald. Indexeddb promised. https://github.com/jakearchibald/idb, 2018 (accessed November 23, 2018).

[27] Lab: Indexeddb. https://developers.google.com/web/ilt/pwa/lab-indexeddb, 2018 (accessed November 29, 2018).

[28] Browser storage limits and eviction criteria. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria#LRU_policy, 2018 (accessed November 29, 2018).

[29] Chris Wilson. Persistent storage. https://developers.google.com/web/updates/2016/06/persistent-storage, 2018 (accessed November 28, 2018).

[30] Dean Hume. Persistent storage api: Building for the offline web. https://deanhume.com/persistent-storage-api-building-for-the-offline-web/, 2017 (accessed November 28, 2018).

[31] Jake Archibald. Introducing background sync. https://developers.google.com/web/updates/2015/12/background-sync, 2018 (accessed November 24, 2018).

[32] Marijn Kruisselbrink Josh Karlin. Web background synchronization. https://wicg.github.io/BackgroundSync/spec/#online, 2018 (accessed November 28, 2018).

[33] Lauren Long. What restful actually means. https://codewords.recurse.com/issues/five/what-restful-actually-means, 2018 (accessed November 27, 2018).

[34] Riemann Hui Joseph Fong. Application of middleware in the three tier client/server database design methodology. *Journal of the Brazilian Computer Society*, (6), 1999.

[35] Prof. Dr. Dr. h. c. Dieter Rombach. Lecture slides of fundamentals of software engineering, 2017.

[36] Peter Zeller. Antidote typescript client. https://antidotedb.github.io/antidote_ts_client/, 2018 (accessed December 06, 2018).

[37] Base64 encoding and decoding. https://developer.mozilla.org/en-US/docs/Web/API/WindowBase64/Base64_encoding_and_decoding, 2018 (accessed December 06, 2018).

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

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

[40] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching Internet

Services with Peer-to-Peer Interactions. *Proceedings of the 26th International Conference on World Wide Web*, 2017. doi: 10.1145/3038912.3052673.

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

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

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

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

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

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

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

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

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

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

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