Department of Computer Science

University of Kaiserslautern

Master Thesis

Offline caching in web applications for AntidoteDB

Server Khalilov

red17electro@gmail.com

University of Kaiserslautern
Department of Computer Science
Software Engineering

Leader:
Prof. Dr. Arnd Poetzsch-Heffter

Supervisor:
Dr. rer. nat. Annette Bieniusa

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Zusammenfassung

Zusammenfassung auf deutsch

Abstract

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

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

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

Acknowledgement

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

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

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

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

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

Kaiserslautern, den 15. December 2018 Server Khalilov

Contents

1 Introduction

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

1.1 Motivation

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

1.2 Research questions

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

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

1.3 The structure

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

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

2 Theoretical background

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

2.1 Main concepts

Distributed database – it is a technology that combines two approaches of data processing: database system and computer network technologies[1]. When we talk about *geo-distributed databases*, however, we mean collections of data that are placed into different geographical locations.

Whenever we would like to interact with such a database, we deal with transactions. A *transaction* is a basic unit of computing, which consits of sequence of operations that are exuted atomically on a database. Transactions transform a consistent database state to another consistent database state, even when they are executed concurrently. As long as each transaction is correct, a database that has a full transaction support 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.

Two other terms we will need are *linearizability* and *serializability*. Linearizability defines a guarantee about single operations on single objects [2]. It guarantees that the same operations are applied in the same order to every copy of the data item[3]. While serializability is a guarantee about transactions, that they are executed serially on every set of the data items[3].

Now, let us introduce different consistency models and the one we will try to follow in the designing of our application.

As stated by Shapiro [4], *strong consistency* model could be described in the following way: whenever the update is performed, everyone knows about it immediately. It means that there is a total order of updates and if there are two different clients that perform the same update, those updates will get some order and everyone will see the same order. The advantage of strong consistency is that it is good for fault tolerance and disadvantages are bad performance and it is problematic to scale.

On the other hand, there are some weaker consistency models that allow better scaling and improve performance. However, it creates problems on the fault tolerance side. In this thesis, we will try to stick with partial *causal consistency*. As it is stated in Zawirski et al. [5], causal consistency is the strongest available and convergent model. They continue their statement saying that under causal consistency, every process observes a monotonically non-decreasing set of updates that includes its own updates, in an order that respects the causality between operations.

2.2 AntidoteDB

For this thesis, one of the core parts in the architecture of the WebCache belongs to the database called AntidoteDB[6]. It is a good choice for the development of correct applications, because it has the same performance and horizontal scalability as AP / NoSQL[7], 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[8]
- delivers updates in a causal order and merges concurrent operations

The last is possible because of Conflict-Free Replicated Datatypes (CRDTs) [9], which is 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. We will cover the topic of CRDTs later in this chapter.

Apart from that, to replicate the data Antidote uses *Cure*[10]. It is a highly scalable protocol, which provides causal consistency, which was described earlier. It lets guarantee that, for example, in social networking applications, a user cannot see the reply to the post before the post itself.

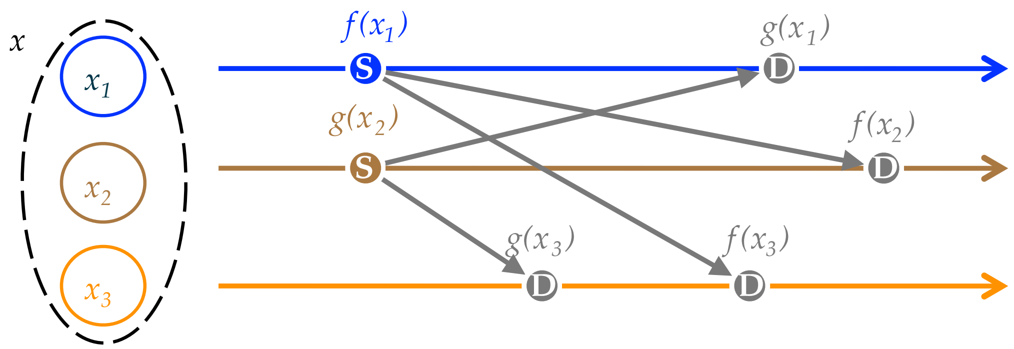
2.3 Conflict-Free Replicated Datatypes

As it is stated in Preguiça et al. [11], a conflict-free replicated datatype (CRDT) is an abstract datatype, which is designed for a possibility to be replicated among replicas and possesses the following properties:

- The data at any replica could be modified independently of other replicas
- Replicas deterministically converge at the same state in case they received the same updates

Replication is a fundamental concept of distributed systems, well studied by the distributed algorithms community[9]. There are two models of replication that are considered: state-based and operation-based. Eventually, we are going to use the operation-based approach. However, we are going to introduce our reader to both of them below.

Operation-based replication approach



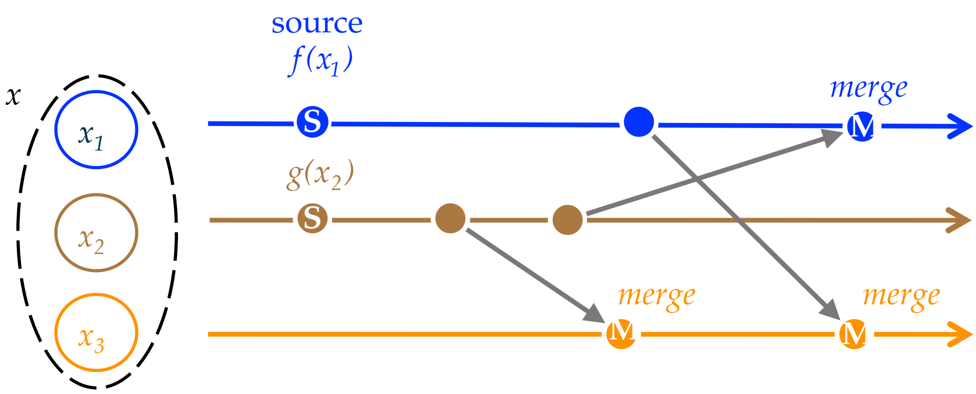
Source: The figure is taken from [9]

Figure 2.1: Operation-based approach.

In this thesis, we are going to use the operation-based replication approach, which originally means that replicas converge by propagating operations to every other replica[11]. Once an operation is received in a replica, it is applied locally. That means, replicas don't exchange states with each other and as sometimes the full state could be a lot of data, it improves the efficiency.

State-based replication approach

The idea of this approach is kind of opposite to the operation-based one. Here, every replica, when receives an update, it first applies it locally. Af-



Source: The figure is taken from [9]

Figure 2.2: State-based approach.

terwards, it sends it to other replicas. That means, that every replica sends its current full state to other replicas. When that happens, the merge function is happening between a local state and a received stated. The important thing is that every update eventually is going to appear at every other replica in the system.

3 Requirements

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

3.1 Problem Description

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

4 Design

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

4.1 Overview of the protocol

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

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

4.1.1 The way of exchaning the data

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

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

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

4.1.2 A client receives an update from the server

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

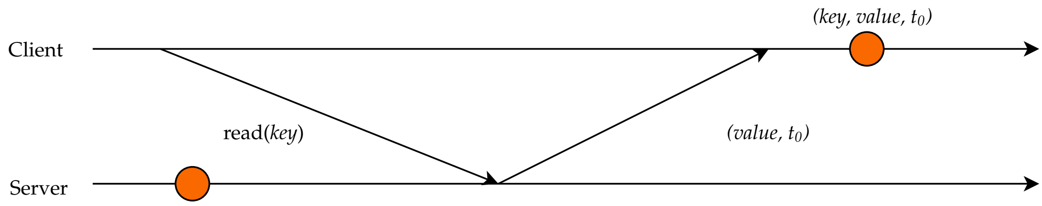


Figure 4.1: The way a client receives a key-value update from the server.

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

4.1.3 A client sends an update to the server

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

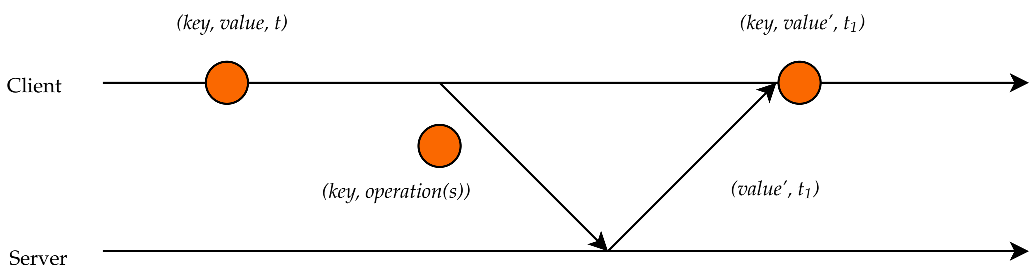


Figure 4.2: The way a client sends a write operation to the server.

Several updates on a client while offline

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

4.1.4 Two clients interact with a server.

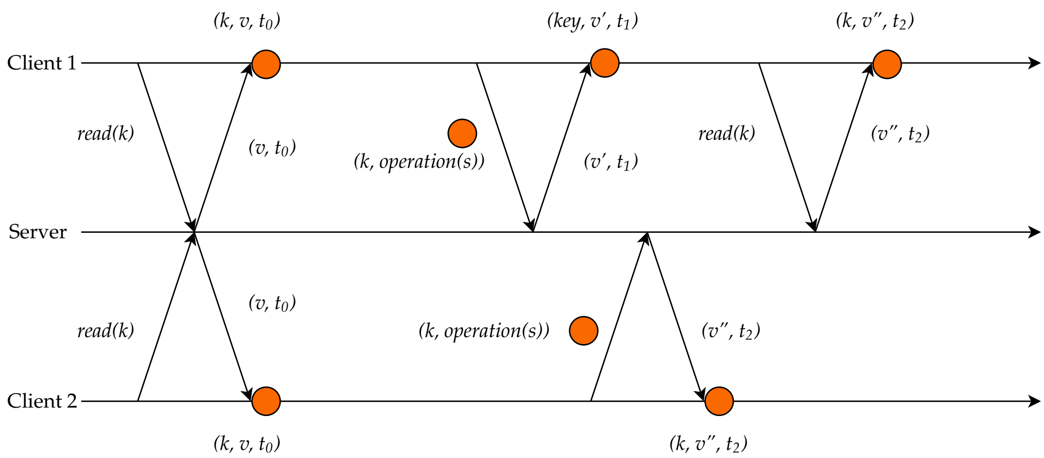


Figure 4.3: Graphical representation over a possible communicaton between two clients and a server.

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

4.2 System's main components

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

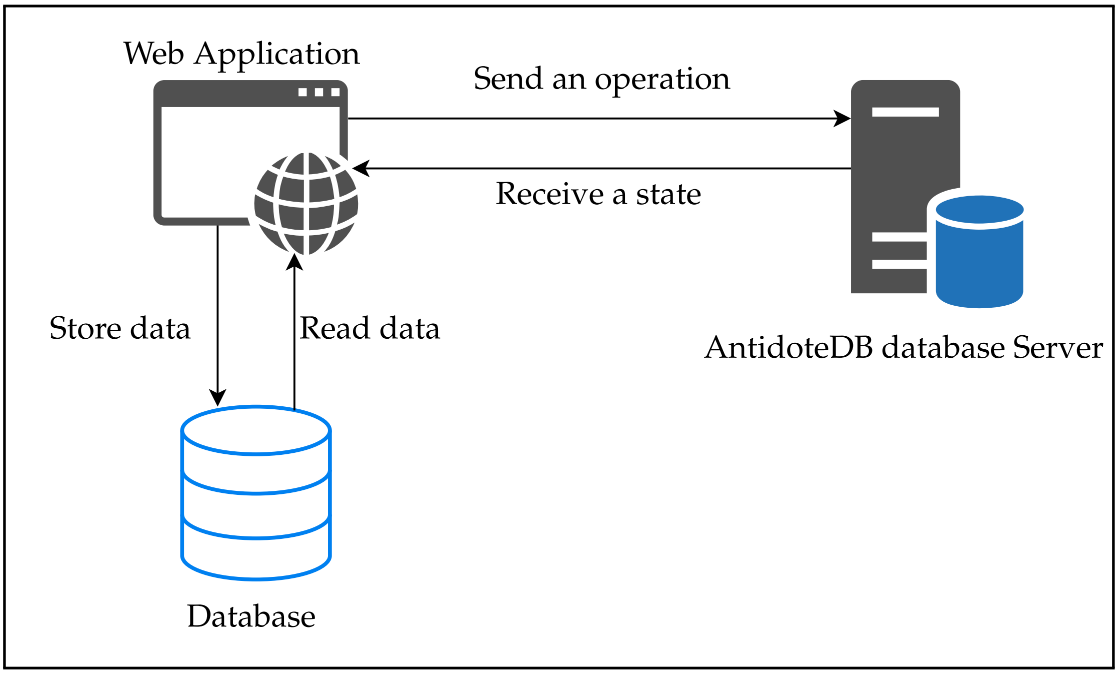


Figure 4.4: A top-level view of the system's design.

4.2.1 Web Application

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

These are the supported commands:

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

4.2.2 Server

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

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

4.2.3 A database layer

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

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

4.3 Offline functionality

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

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

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

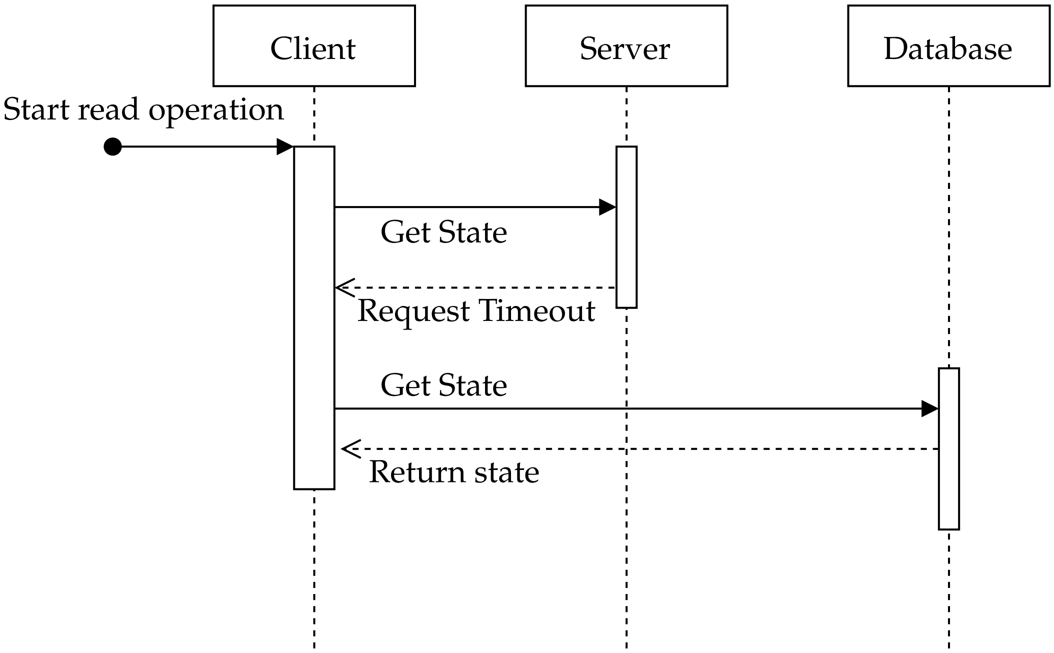


Figure 4.5: Successful request of state from the local database – Sequence diagram.

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

4.4 Online functionality

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

4.5 The transition between offline and online modes

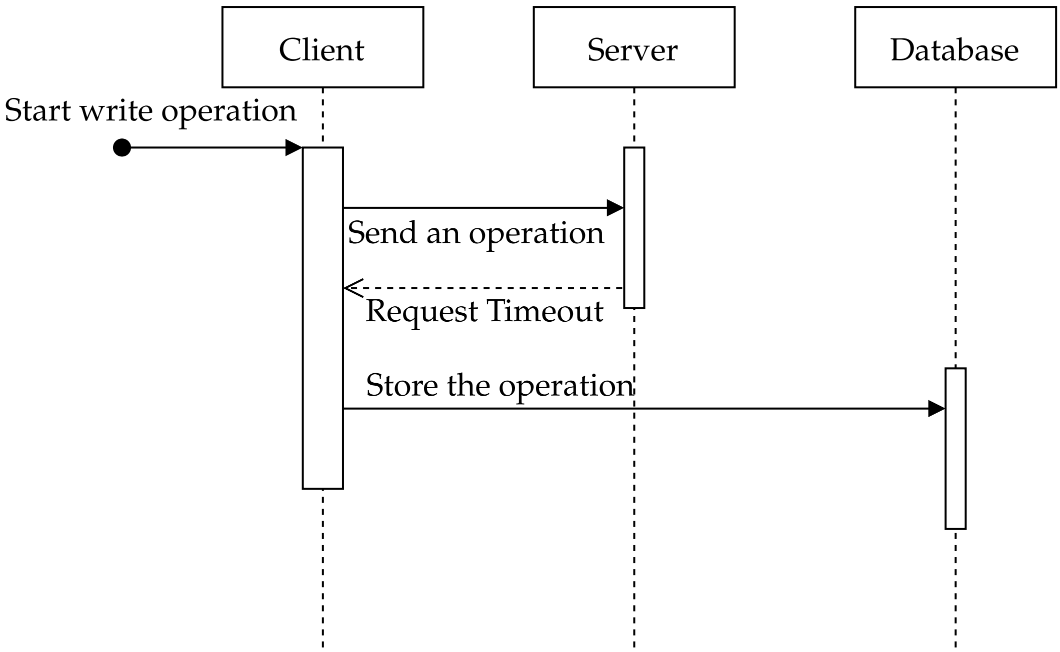


Figure 4.6: Successful storing of an operation in the local database – Sequence diagram.

5 Technologies

This chapter consists of detailed description of used technologies to accomplish thesis goal. I believe that the implementation of the web-client for the AntidoteDB with a cache support would require the following components: ‘

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

5.1 Service Workers

A service worker[12] is a JavaScript file that sits between the application and network requests. As well as that, it runs separately from the page. It has the following properties:

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

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

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

5.2 IndexDB database

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

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

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

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

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

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

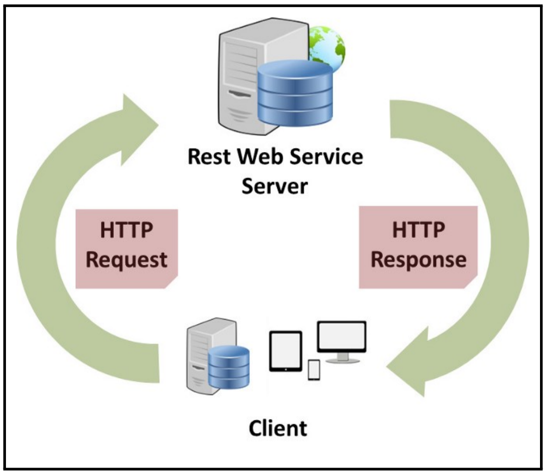
5.3 Cache API

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

6 Implementation

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

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



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

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

6.1 Optimization

7 Evaluation

Evaluation

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

8 Related Work

In the following paragraphs. we are going to mention the researches, which influenced our work.

Geo-replication of data into several data centers (DC) across the world is used in cloud platforms in order to improve availability and latency[13]. 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[14]. The authors of the SwiftCloud state that it improves latency and throughput if it is compared to general geo-replication techniques. That is possible to due to availability during faults thanks to automatic switch of DC when the current one does not respond. Apart from that, the system also maintains consistency guarantees [5].

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[15]. 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 clients. The guarantee of convergence between all of the replicas is possible due to CRDTs.

9 Conclusion

Conclusion

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

9.1 Summary

9.2 Future Work

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

Idempotence of updates

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

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

There are possibilities:

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

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

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

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

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

List of Figures

List of Tables

List of Code

Bibliography

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

[2] Peter Bailis. Linearizability versus serializability. http://www.bailis.org/blog/linearizability-versus-serializability/, 2014 (accessed October 24, 2018).

[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] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.