Draft of the Architecture for the AntidoteDB Web-Client with Cache support and Offline-first experience

1. Introduction

1.1. Research Questions

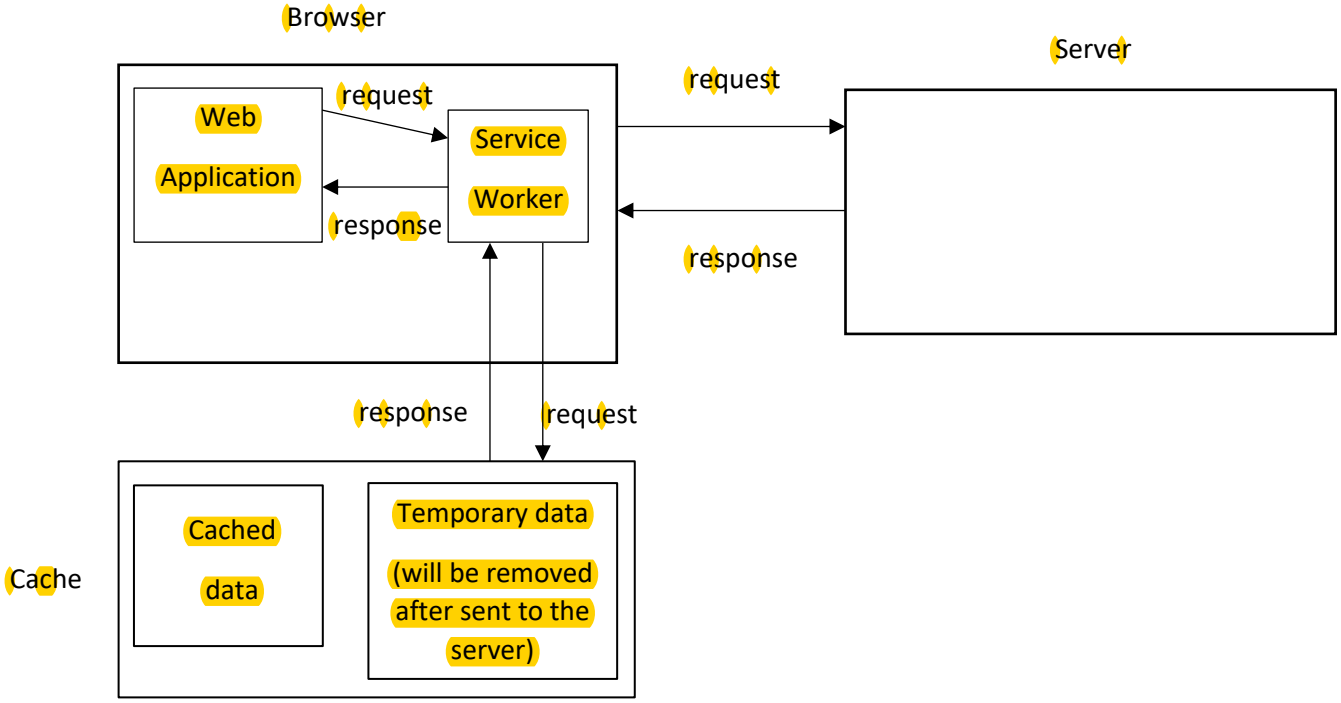
RQ1. How efficient is it to use a web-client with cache rather than without it?

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

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

2. Architecture

2.1. Architecture Overview



I believe that the implementation of offline-first experience for the AntidoteDB Web-Client supporting cache would require the following components:

- A service worker (to control the requests coming from the client).
- A cache API (to store assets information. Might be unnecessary, though).
- A database (for storing some specific key-value data)

2.2. Service worker

A service worker is a JavaScript file that sits between the application and network requests. As well as that, it runs separately from the page. Properties of it:

- 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 network traffic. In situations, when there are problems with network, we are going to get content from cache (which consists 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, 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.

2.3. In-browser 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 bypasses both the service worker and the HTTP cache), and we start receiving new data records. When we receive 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 use 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 an object store 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 successful:

  In this case, the request would end up changing the data on the server 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 successful:

  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 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.

This is a very short description of the protocol and several problems could arise. It is going to be discussed later in a separate section for the protocol description.

2.4. Cache-API

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

3. The description of the protocol for the connection of a web-client and Antidote DB-based server

3.1. Introduction

In order to specify the protocol, we will investigate the following cases:

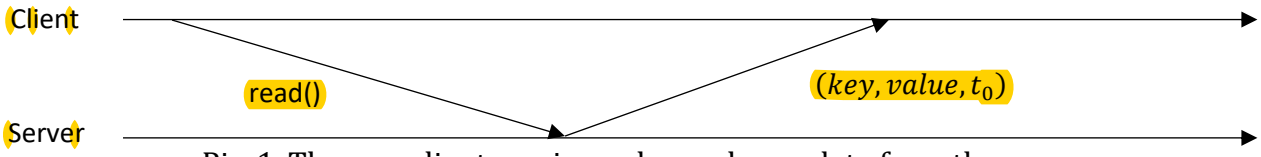
- A client receives updates from the server.
- A client sends updates to the server.
- Two clients interact with a server.

As there are different possibilities to exchange the updates (a state, a list of operations or the values), we will consider the exchange of key-value pairs (k, v) with a specific timestamp t, identified on a server part.

3.2. Description

3.2.1. A client receives updates from the server.

Let's 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 key-value pair and the timestamp t_0.

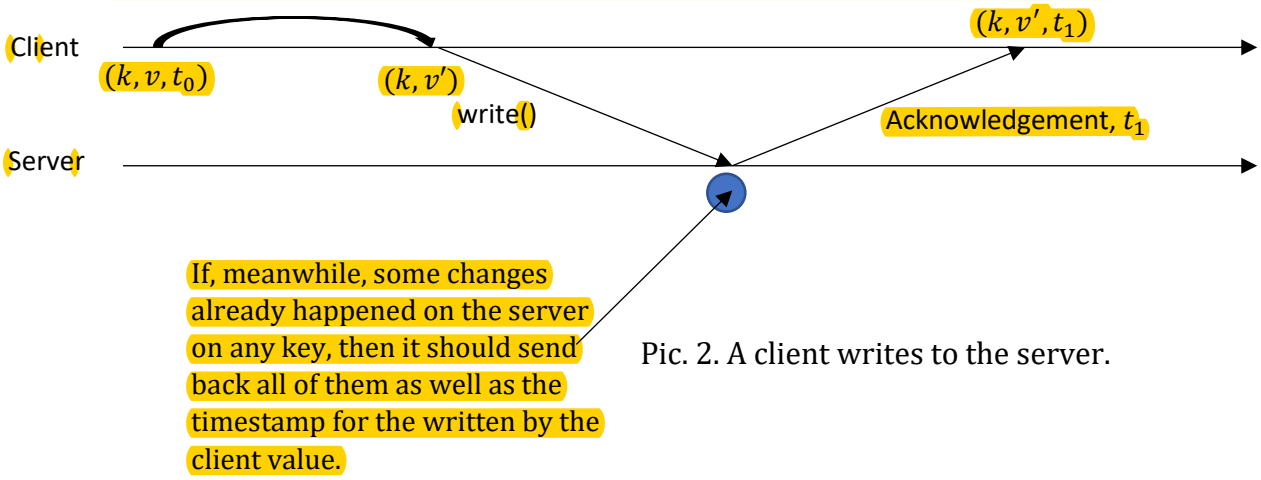


Pic. 1. The way client receives a key-value update from the server.

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 continues waiting for fresher updates.

3.2.2. A client sends updates to the server.

In the case of writing the information to the server, a client has to send a key-value pair to the server and if the operation performed successfully, then the acknowledgement with a timestamp is going to be sent back to the client:



Pic. 2. A client writes to the server.

Several updates on a client while offline

In this case, when the connection is established again, the list of updates on the client should be sent as an array to the server once at a time.

Idempotence of updates

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 doesn't know which of these situations happened. Therefore, we have to find a general solution for such kind of behavior.

1st possible solution:

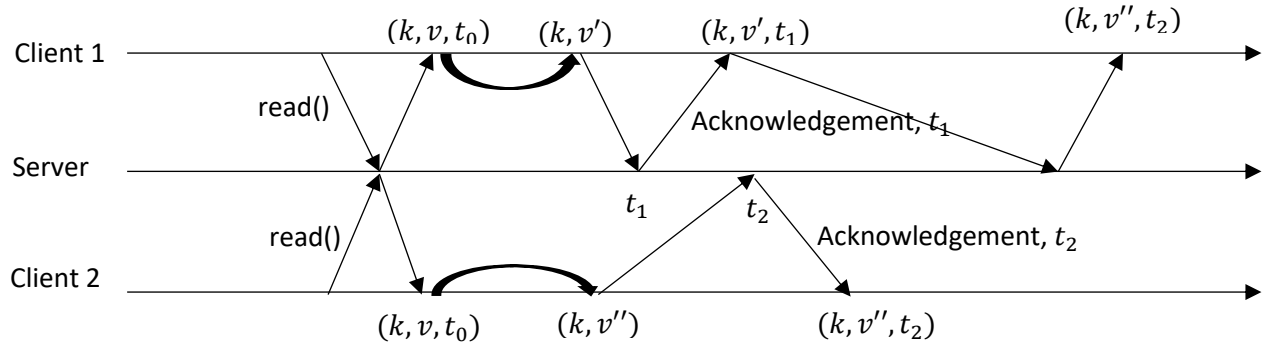
- It does not matter, whether the update was applied or not. A client will just send an update again, till it does not receive an acknowledgement, regardless of what happens on the server's side.

2nd possible solution:

- 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, 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 improvements will be discussed.

3.2.3. Two clients interact with a server.



Pic. 3. Graphical representation over a possible communication between two clients and a server.

Let's 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 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.

3.3. Afterthought

The communication approach described above could use a JSON format in order to transmit objects consisting key-value pairs.

Moreover, the described protocol would allow us to handle a CRDT-datatype separately on a client and a server, because the communication would consist only of key-value pairs.