PAVOL JOZEF ŠAFÁRIK UNIVERSITY IN KOŠICE
FACULTY OF SCIENCE

DOCUMENT STATE SYNCHRONIZATION WITH OFFLINE MODE

2018/2019 **Šimon Kocúrek**

PAVOL JOZEF ŠAFÁRIK UNIVERSITY IN KOŠICE
FACULTY OF SCIENCE

DOCUMENT STATE SYNCHRONIZATION WITH OFFLINE MODE

BACHELOR'S THESIS

Study programme:	Informatics
Study field:	9.2.9. -- Applied Informatics
Workplace:	Institute of Computer Science
Thesis supervisor:	RNDr. Peter Gurský, PhD.
Thesis consultant:	Mgr. Martin Večeřa

Košice 2009 **ŠIMON KOCÚREK**

Univerzita P. J. Šafárika v Košiciach
Prírodovedecká fakulta

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Šimon Kocúrek
Študijný program: Aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.9. aplikovaná informatika
Typ záverečnej práce: Bakalárska práca
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Document state synchronization with offline mode

Názov SK: Synchronizácia stavu dokumentov s off-line módom

Cieľ:
1. Analyze current document state synchronization solutions
2. Compare and choose most fitting synchronization, storage and communication methods with it's implementation suitable for web browser solution
3. Enrich the existing document state synchronization implementation with offline capabilities

Literatúra:
1. Benson E. Marcus A. Karger D. & Madden S. (2010 April). Sync kit: a persistent client-side database caching toolkit for data intensive websites. In Proceedings of the 19th international conference on World wide web (pp. 121-130). ACM.
2. Neil Fraser (2009 January), Differential Synchronization. In Proceedings of the 2009 ACM Symposium on Document Engineering (pp. 13-20).

Vedúci: RNDr. Peter Gurský, PhD.
Konzultant: Mgr. Martin Večeřa
Ústav : ÚINF - Ústav informatiky
Riaditeľ ústavu: prof. RNDr. Viliam Geffert, DrSc.

Dátum schválenia: 03.05.2019

Univerzita Pavla Jozefa Šafárika v Košiciach
Prírodovedecká fakulta
Ústav informatiky

Abstract

In this paper we focus on solving the issues caused by going offline inside a state synchronizing network. In order to keep the state consistent across all clients in a network, various synchronization methods are used. These methods expect all clients to be available at any time and don't describe situations where some of the clients go offline and later attempt to reconnect. First, we analyze a sample of synchronization methods, while comparing the ways they could support offline mode. After that we choose the most fitting method and provide a detailed description of possible modifications allowing client re-connections without loss of data or state conflicts. Finally, we implement and compare these modifications.

Key words: Synchronization • Offline • Implementation

Abstrakt

V tejto práci sa zaoberáme riešením problémov spôsobených odpojením klienta zo siete, ktorá synchronizuje stav klientov. Na udržanie konzistentného stavu naprieč všetkými klientami v sieti sa používajú rôzne synchronizačné metódy. Tieto metódy očakávajú nepretržitú dostupnosť všetkých klientov a nepopisujú situácie, kde sa niekoľko klientov odpojí a neskôr pokúsi opätovne pripojiť. Najprv zanalyzujeme vzorku synchronizačných metód, porovnávajúc spôsoby, akými by mohli podporovať offline mód. Následne vyberieme najvhodnejšiu metódu a poskytneme detailný popis možných úprav umožňujúcich znovu-pripojenie bez straty dát, alebo konfliktov. Nakoniec implementujeme a porovnáme tieto úpravy.

Kľúčové slová: Synchronizácia • Offline • Implementácia

Table of Contents

Terminology

Document is a written piece that trains a line of thought. For our purposes we will consider a document any single block of binary or text data, stored in an enriched text format.

Synchronization refers to a process of establishing consistency among data from a source to a target data storage and vice versa. This is more commonly known as data synchronization to prevent confusion with process synchronization.

State of a program is a serialized version of variables and constants of a program at a specified time. Any two processes in the same state must behave identically upon receiving identical inputs.

Offline indicates a state of disconnection from a network, as opposed to indicating being in a ready to user state. In this thesis we consider being offline as being disconnected from the network our server is located in, rather than being disconnected from any network.

Reconnection is an event, where client regains connected state to the server and therefore becomes online. This even can happen only for clients already situated in an offline state.

Collaborative System is an application software designed to allow people involved in a common task to achieve their goals. We will consider collaborative system any system, that allows real-time collaborative editing of at least one document in a web application. This editing is performed collectively by multiple clients, where each edit is propagated to all clients with real-time feedback.

Overview of the chapters

In the introduction chapter we start by defining the problem at hand. We explain, why is the lack of offline mode an issue and who it affects. With this we also conclude our motivation to implement this offline mode solution, while describing the benefits it brings. We also provide a brief overview of the current state of the solutions to this problem.

Next, we dig deeper into existing solutions. We show what features they have, lack and how they approach problems we defined. We will use these findings as an inspiration during our implementation. For each implementation we will mention what how will our solution improve upon its concept.

In the third chapter we focus on the way we approach the larger problem of implementing the offline mode. We break it up into smaller subproblems and describe each subproblem with challenges it presents, possible solutions with their pros and cons of using the solution in an offline mode implementation.

The Defined architecture chapter servers as an implementation guide for creating document state synchronization offline mode. For each defined subproblem from the previous chapter it describes our solution and its possible implementation. After that we provide a chapter that briefly goes over code samples that were crucial to our solution.

The last two chapters describe characteristics of our custom implementation of offline mode for document state synchronization. They provide details of choices we made during implementation together with time and space complexity of our solution. Also mentioned are possible improvements and alternatives with their tradeoffs. Besides possible improvements, we will describe any shortcomings of our implementation, that might be improved upon by modifying, or extending our solution.

Introduction

Have you ever been in a situation, where the only network connection you had was flaky at best, with low speeds and frequent disconnects? This type of connections is often the only option in third world countries and is common even in countries with developed internet infrastructure.

Such flaky connection can cause more than annoyance with pages not loading fast enough. Many people working in healthcare, finance or government spheres require internet connection to perform their jobs.

For these people sudden disconnection might mean loss of important work. When connection to the server is lost, collaboration systems often discard all work that wasn't uploaded. Leading to important meeting invitations not being sent, or patient health statements not being recorded.

Often the collaboration systems disregard this as an issue and expect users to use stable internet connection instead. While not without tradeoffs, implementing offline mode is feasible task, that is also far simpler than implementing collaboration system.

Our goal is therefore to solve this problem by providing a solution for offline mode in collaborative systems. A solution that would remember all actions performed by user after going offline and upload this whenever possible.

Offline would lead to better user experience among all users, regardless of their internet connection. While also allowing more reliable web applications, that can be used in sensitive environments, without the fear of losing data.

1 Why is Offline Mode Needed?

The problem of unreliable network connection is something that website users experience in both urban and rural areas no matter if they live in first or third world country. Besides the inconvenience and frustration, it causes it can have more serious effects if it prevents people from performing their jobs, or straight up leads to them losing the work they have done.

This might not seem like a big deal if we are talking about personal use of applications, however for collaborative systems used in healthcare, or finance, being able to use applications in every case could potentially save lives or prevent companies from going bankrupt.

Offline mode for web applications is a solution that would help all users relying on the application for commercial or personal purposes. Not only would it allow for easier collaboration where stable internet connection is no longer a necessity, this mode would also lead to greatly improved user experience.

1.1 How Should an Offline Mode Work

To prevent all issues with losing network connection, our fist main concern is allowing user to work even when they can no longer submit their changes to the server. This is crucial for an offline mode implementation, as without this no work can be done.

The second main concern of our solution is how the event of reconnection is handled. It is unacceptable that any work that was previously stored on the server or that was performed by the client while offline is lost. All progress must be merged manually by users or stored somewhere so that it is available for use or later merging.

For the user experience of applications that support offline mode to not be hindered, the application still needs to work in real-time. That means that all changes that the system never stops, or locks down. This should not happen even in case larger conflicts need to be merged.

Offline mode for collaborative systems should be an abstract concept, that isn't proprietary or tied to a specific technology stack. For this reason, we only consider a stand-alone library, or extension of existing library for document state synchronization an appropriate solution to the problem.

Similarly, the offline mode should not introduce new behavior to the already existing synchronization system and should be able to work on a single document, rather than use any extensions like file system synchronization to perform reconnection merge.

Our offline mode also needs to work in multiple network scenarios. It is not enough that the mode would only work in cases where server communication is impossible for a time period longer than 10 minutes, in which significant state changes were introduced on both client and server side. A situation, where disconnections are frequent, but short should also be considered and handled with already mentioned characteristics.

2 Current State of Offline Synchronization

To battle the issue of frequently disconnecting clients and improve user experience, web browser vendors created many mechanisms, that made way for websites usable without internet connection. These allow web developers persisting data between user sessions and running code before the webpage gets loaded.

With time, applications utilizing this offline capability, are becoming more and more popular. This raise in popularity is partially caused by the progressive web application movement, which states offline mode as one of required parameters of every progressive web application.

Progressive web applications are web applications, that can be used in same way as native applications. They utilize offline storage, notification APIs and service workers, to provide better user experience, more functionality and lower network usage. Currently their only drawback being browser support, where currently only one of the major browser vendors supports all APIs needed to implement such application.

The acronym API stands for Application programming interface. In the context of offline storage mechanisms, API refers to a set of implementation independent functions, with their functionality specified by the W3C (Hickson, 2016).

Parallel to the rise of progressive web applications, is the increase in popularity of complex single page applications, utilizing modern JavaScript frameworks. Single page applications allow webpages to have more complex logic, actively respond to user actions and communicate with server without refreshing the page.

2.1 Existing Solutions and Their Drawbacks

With wide browser support for complex web applications with offline capabilities, offline modes for some collaboration systems appeared. Each system described further chose a different approach to implementing the offline mode. We will describe the pros, cons and inner workings of this approach and show, why this solution does not meet all

requirements for a collaboration system offline mode that does not hinder user experience.

2.1.1 Office 365

The first example of a web application that supports collaboration and is vital for many professions is Office 365 made by Microsoft. Historically this tool did not support multiuser collaboration and was based on a locking mechanism, where each user locked his file, preventing others from modifying its content.

Nowadays it is possible for multiple users to edit the same document, with real-time feedback. However, problems start to appear once internet connection is lost. The application gets stuck in a state where “Saving…” dialog is displayed, and user is free to continue working.

The main issue with this system is that the saving dialog never disappears, and user state is not saved even after regaining connection. This is not ideal, because it gives user the illusion that his work will eventually be saved, when there is no system for reconnections in place.

2.1.2 Google Docs Office Suite

Another widely used office suite is Google Docs, made by Google. This office suit has similar capabilities to those of Office 365 and like Office 365, Google Docs is also integrated with its own file storage for storing documents.

With multiuser collaboration implemented by integrating Operational Transformation synchronization method this application serves as an example that robust collaborative applications can be built using this method without any issues.

When it comes to offline capabilities, Google Docs can be considered a direct improvement over Office 365. After user loses internet connection, this event is quickly discovered and user is notified that his progress will no longer be saved, until he reconnects.

To prevent the need for synchronization upon reconnection, the application locks entire document and user is not able to perform any change while offline. While not ideal, as we would like that users are still capable of working even when offline, this system prevents the user from doing work that would be eventually lost anyway.

The preferred solution Google created for offline mode is a custom browser extension that allows working even without connection to the server. For our purpose this is a very cumbersome solution as creating a browser extension for every supported browser requires a lot of work. Offline mode should be part of the synchronization and not additional application tied to the browser the user is running.

2.1.3 Visual Studio Live Share

New addition to collaborative systems are text editors using CRDTs for state synchronization. The biggest difference between these systems and more traditional systems based on Operational Transformation method is that server is not required. For this reason, we redefine a disconnected client as a client that is unable to reach all other clients in the network.

While peer to peer networking allows for a higher tolerance of network partitions, Visual Studio Live Share does not take advantage of this feature. In case of disconnection the client session is terminated, meaning he will no longer be able to see documents of other clients and his documents will no longer be shared.

This sounds like a reasonable implementation of offline mode; however, this fully prevents all clients from editing anything other than their own documents while offline, making it unusable as an offline mode for collaboration.

2.1.4 Box

Box is a file sharing service, which serves as an alternative to cloud storage services like Dropbox and Google Drive. While its collaboration capabilities are very limited[1], the offline mode approach is quite unique.

[1] Box file collaboration works by locking files, like old Office 365 versions, while also keeping change history, to allow undoing unwanted changes.

In case user was editing a file and went offline a notification is displayed, while user can continue doing his work on opened file. During this time, his lock is released, and others are free to edit the file concurrently. After reconnecting the client version is saved as a copy of the edited file, ensuring that both server and client state are preserved.

While original, this approach has few limitations. First, in a collaborative system that does not support multiple files, creating such offline mode would require adding additional functionality to the original collaborative system. Another minor limitation is that the states are not merged upon reconnection, leading to multiple users each editing their own file, instead of all users collaborating on the same file.

2.1.5 Meteor Framework

Meteor is a JavaScript framework with offline mode support. Applications written using Meteor should have offline capabilities by default, resulting in an offline mode for any web application, even ones that allow collaboration.

It creates this offline mode by performing optimistic updates and simulating server response even when there is none. When offline all requests and data gets stored[2] and are persisted even after user session ends. Persistence of this kind is a nice to have feature for collaboration system offline mode, as it guarantees that no work is lost even when user reloads or closes the web application.

The problem of this solution lies in its generality. Collaborative systems require special treatment during reconnection, which is something Meteor does not provide. All server requests and responses that are a cause of conflicts are straight up ignored, without any merging.

Another and main issue of this framework is that it ties the developers to a specific proprietary technology stack, where Meteor requires that it runs both on the frontend and the backend and therefore is not worthy for a general state synchronizing offline mode.

[2] This storage might require usage of a third-party library created for this framework.

2.1.6 Git Version Control System

Currently Git is the most widely used version control system available. While not a collaborative system by definition, its main purpose is to allow collaboration of multiple users on same data. It does so by sacrificing the real-time aspect of collaborative systems. All clients are expected to work offline and submit changes only once they are done.

Git works fully offline; inspired by its approach to state synchronization we can create a custom offline mode solution that would also work in real-time and therefore be an offline mode for fully fledged collaborative system.

Git's works in a similar way to Differential Synchronization (See chapter 3.2.2.4) with the only difference being that merging is not done automatically if there are conflicts. Before submitting changes to the server all conflicts need to be resolved manually by the user.

3 Approach to Implementing Offline Mode

Before we start with implementation of an offline mode for a state synchronizing network. We first analyze and compare current document state synchronization solutions, with primary focus being differences in synchronization methods and the way they resolve state conflicts.

Next, we will shift our focus towards implementation and describe advantages and disadvantages of possible offline mode implementation for each considered synchronization method.

As we plan on creating a solution for web applications, we will take into consideration web browser limitations as well as feature support. With this in mind, we will choose the most fitting storage method for saving any progress made while offline.

And to allow push notifications from the server, needed for state change propagation to all clients, we will also describe possible solutions alongside pros and cons of using them during client reconnection.

After we are done with the analysis of all theoretical and technical aspects of our solution. We enrich one of the existing, open source document state synchronization implementations with offline capabilities.

3.1 Problem Definition

All web applications are based on a client-server model, where server is given by a Universal Resource Locator[3] and the client is user using a web browser. This fact gives us a flexibility in interactions between users, in case they are all connected to the same server. Allowing us to implement a collaborative system inside a web application.

Client-server model is a distributed application structure where the server is a service provider, responding to multiple clients, that are using the provided service. This

[3] More commonly known as the URL.

model is an alternative to less used distributed Peer-to-Peer model. Client-Server model is centralized, with server being the center of communication, which often leads to simpler implementations of stateful systems and systems with authentication.

Where this model comes into play, is collaborative system implementation. It allows simple propagation of changes through a common server. In order to commit a performed edit, client only needs to send one request to the server, which later propagates the change to all other clients.

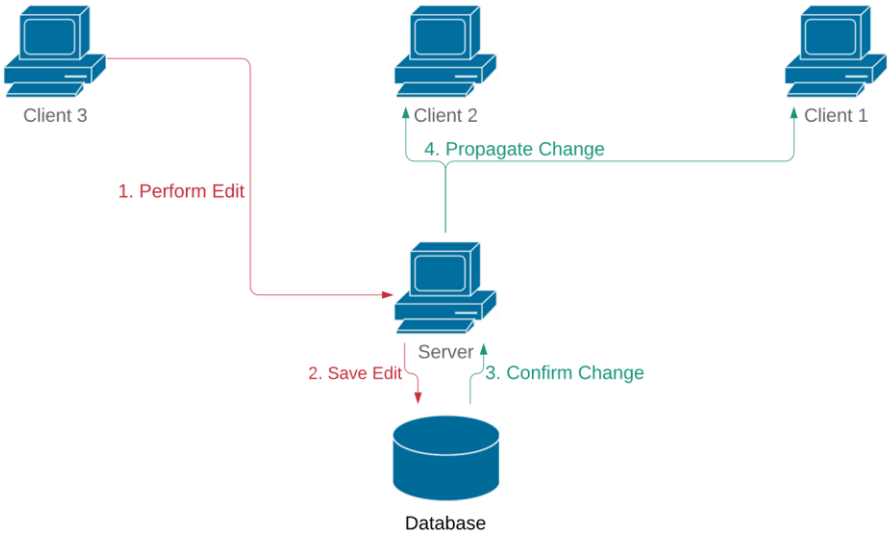

Figure 1 Client change propagation

The server has multiple roles in this scenario:

- It serves as a storage for the data all clients operate on, meaning all that is needed to change the data is sending a request to the server. Which stores the data in a database, before further propagation.
- The server handles propagation of each change performed on the data to all clients, so that every client operates on the changed data, rather than the old one and state is consistent across all clients.
- Conflict resolution, as well as any transformations of the received data are handled by the server. Allowing for simpler client-side implementations and more secure systems.
- Allows checking for unauthorized requests and constraint violations, since the server can serve as an authority to authenticate against.

The already mentioned conflict resolution is a mechanism, for handling state conflicts. These occur in case when data changes performed by two or more clients are mutually exclusive. Example of such mutually exclusive actions are editing and deleting same paragraph of text or adding different words on the same position in text.

In the case of state conflicts, it is not clear how the two different states should be merged together. Currently there is no right, or definitive solution, as the merging depends on the states being synchronized as well as system preferences (Korłub, 2017).

It is desirable that users aren't bothered with small conflicts, even when it means generating corrupted data, or throwing away changes made by some user. As users can repeat the previously performed action again, without any concerns.

When larger state conflicts occur, any possible data loss is unacceptable. We tackle this problem, as it’s essential that user’s work isn’t lost after reconnecting, resulting in a situation similar to losing work in a synchronization system without offline mode.

Handling large state conflicts, together with implementation of storing data, while disconnected, will result in an offline mode implementation, where client separated from the server in the event of network partition[4] will be able to synchronize his state with the other clients when connecting to the server becomes possible again.

3.2 Subproblems Encountered During Implementation

During this process of analyzing and implementing our offline mode solution we noticed that our problem can be divided into multiple smaller subproblems, that, when implemented correctly, would result in a correctly implemented offline mode document state synchronization.

[4] Situation, where network is divided into mutually unreachable subnetworks.

3.2.1 Detecting a Network Partition

Early on the first such subproblem, we discovered was detection of network partition. In order to activate the offline mode, we need to know that such even has occurred, and online mode would result only in sending failed request to the unavailable server.

Upon detecting the event of network partition, the process of starting offline mode will be started. This process requires saving all data necessary to restore same state even after a page refresh.

The data that needs to be saved consists of all requests that weren't sent yet and are in a pending state as well as all requests that were sent and now are waiting for a response from the server. Only once all these requests were safely persisted, we can start the offline mode, where every change is stored locally.

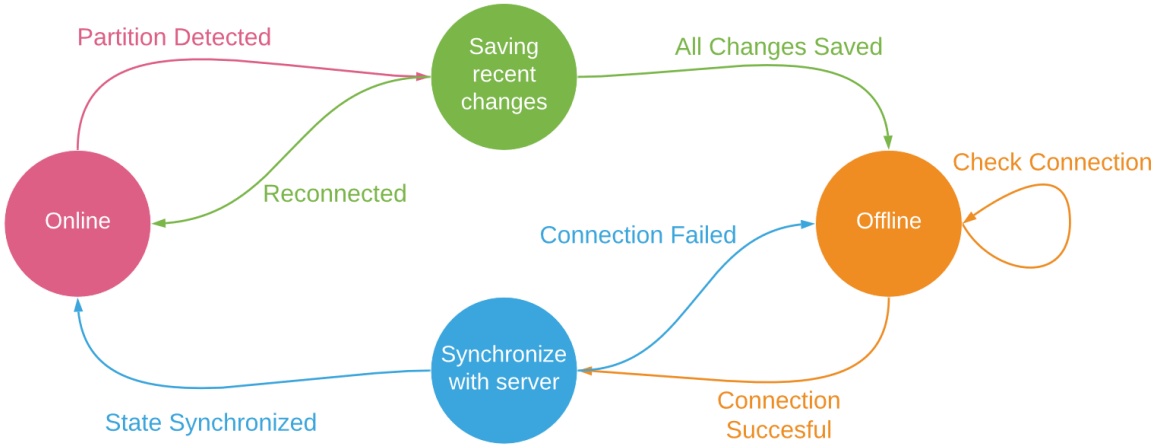

Figure 2 Offline mode state diagram

There are several ways we can detect a network partition. As all methods achieve the required result that's needed for the offline mode implementation, we will keep our focus on the efficiency and handling of edge cases while comparing them.

3.2.1.1 Heartbeat

The first way of detecting that server is no longer reachable is a heartbeat. This method works by sending a periodic request, usually in order of seconds to the server, to

confirm that response is received. In case no response from server is received at for a heartbeat request, we can assume that network partition has occurred.

This solution is often employed in distributed network environments, where each network node sends heartbeats to a dedicated master, or registry server keeping track of working nodes. The main advantage this provides is that it is possible to apply auto healing before the unavailable node is required.

However, the main disadvantage of heartbeat is that it requires additional network traffic, in form of periodically sent request. And the fact that we can't perform any auto healing in case the unreachable clients are users, other method would be preferable.

3.2.1.2 Browser Callback

Another seemingly correct approach, that would not result in an increased internet communication is using the built-in browser events for connecting and disconnecting from the network:

```
window.addEventListener('online', function() {}, false);
window.addEventListener('offline', function() {}, false);
```

This solution is rather naïve as it does not detect all possible cases when user goes offline. The event itself is implemented differently on each browser, with one common pattern. All browsers use some properties or mechanisms of operating systems to figure out if network interface is up, or internet connection is in some way enabled.

This however does not mean that server is reachable. There are numerous cases when this event fails to trigger. In all these cases we need to start the offline mode, as none of the performed changes reach the server:

- Client is connected to a network that is not a part of internet network.
- Client is in a network that has server's port or IP blocked.
- Network partition occurred along the way to the server and therefore server is unreachable from the client's network.
- Client has lost internet connection; however, his operating system has failed to register the event yet.

- Server has disconnected from the internet, making it unreachable, while client still has stable internet connection.

While very simple to implement and supported by all major browser vendors, due to the number of edge cases when network partition would not be detected, we decided to ignore this option completely.

3.2.1.3 Request Timeout

Another possible solution is utilizing the request timeout feature. This feature starts a timer for each request sent to the server. If a successful[5] response is not received in the requested timeframe we conclude that the network partition has occurred, and we need to start the offline mode.

From the start we can see that this solution would not increase the network traffic, as it never sends any additional requests, besides the ones the client would send anyway. Additionally, no edge cases where network partition would not be detected are not possible, as not being able to communicate with the server is what defines network partition in our case.

For our use in web application with offline mode, this solution is superior to the previously suggested ones. The only disadvantage this solution presents is that it does not detect the partition as it happens, but only after the client tries to communicate with the server.

This disadvantage is not an issue, as we need a transparent solution that works without the user having to react to the event of going offline anyway. Additionally, it guarantees us that at the time of enabling offline mode, there will be at least one request in a pending state, waiting to be sent to the server, that needs to be persisted for offline use.

[5] It is possible to receive a response even in case the server is unreachable. However, this response could be sent by routers on the way to the server, rather than by the server itself. The point of this response is to indicate what issue is causing the network partition and therefore is considered an error response.

3.2.1.4 Detecting Reconnection

Like the problem of detecting network partitions, we also need to detect that server is reachable from the client once again. Once that happens, we can fully synchronize the local state with the server. If this synchronization finishes successfully, we can transition back to the online mode.

Unfortunately, unlike the detection that network was partitioned, we don't have many options here. The only reliable way to find out that server is by polling. Polling works similar to the heartbeat in a way, that it periodically sends ping requests to the server in a specified time interval.

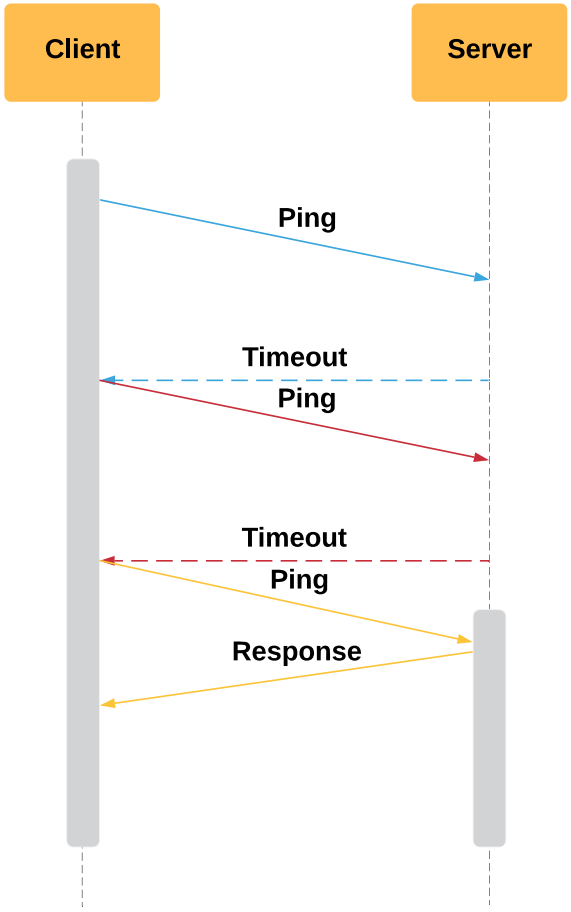

Figure 3 Reconnection polling

The seeming issue with this approach is increased network traffic. However, most of the time this polling is used is in cases, when client has no internet connection. Sending a request in a computer with no internet connection has no real impact on the internet traffic and only minor effect on the client.

Even when sending a request is a cheap operation, it can be potentially harmful on devices powered by batteries. Periodically sending a request means waking the processor and network card up, both of which consume large amounts of power.

To improve on this, we can use the fact that once user loses connection and don't reconnect in a short time interval, it usually means they will be offline for a longer time and therefore we can increase the time between polling requests.

Instead of having a constant timeout and sending a new request each time response is not received in that timeframe, we can double the timeout and therefore make requests less frequent, when we expect client to be offline for a longer time.

Since the timeout doubles each time, we can quickly end up in a situation, where client sends only one request a day to check if connecting to the server is possible. To prevent this unwanted issue, we would also need to set up some upper bound for the timeouts, such that sending frequent requests is not wasting battery power but is still able to reconnect in a reasonable time.

3.2.2 Choosing the Right Synchronization Method

When implementing collaboration systems, we can use an already implemented library for synchronizing state across multiple clients. There are many options to choose from, with the biggest differences in a collaborative system behavior coming from the synchronization method that was used.

These methods describe how the state should be stored, what messages are sent to the server and how to handle conflicts. As they are academically studied and proven to work, they can serve as an abstraction we will base our offline mode upon.

3.2.2.1 Document Locking

The simplest possible way to keep state consistent among multiple users is by locking the document. In its most basic form, this technique allows only one user to edit the document at a time, while others have a read-only access to the document. This basic

form cannot be used to implement a collaborative system, as, by the definition, multiple clients must be able to collaboratively edit the document at the same time.

To fix this issue, the document can be split into multiple smaller parts, each with its own lock. With such fine grade locking, we could implement a collaborative system, that would be conflict free, as no two users would be able to edit the same locked part of the document at the same time.

However, after introducing this fine grade locking for subsections of the document, we would lose the simplicity and introduce a new problem. In an environment with unstable network connection, lock requests and releases could be potentially lost, resulting in a client attempting to edit subsection he has no lock for, or worse, subsections of the document remaining locked, with no client having the lock.

Additionally, while this method provides a simple mechanism to ensure consistent state among users, it has not mechanism for merging state after reconnection. Leaving all the implementation on the developers. Combined with the instability of this method when using multiple locks for document subsections, it is unsuitable choice for our offline mode implementation.

3.2.2.2 Conflict Free Replicated Data Types

Conflict Free Replicated Data Types, or CRDT for short are described in a paper by Marc Shapiro, Nuno Preguiça, Carlos Baquero and Marek Zawirski (Shapiro, et al., 2011). The CRDT is the newest addition in the field of state synchronization. Despite it being relatively new and unproven, it got quickly adopted by many databases, chat systems and text editors due to its simplicity and ability to synchronize state without any central server.

This ability to synchronize without server is the biggest advantage CRDT has over the older synchronization solutions. It allows synchronization in Peer-to-Peer applications and in distributed networks, where all clients need to share data between each other (Smith, 2016).

Data synchronized using CRDT is guaranteed to be eventually consistent, meaning that, given enough time, all clients will arrive at a consistent state, where regardless what server state is queried, consistent response is returned every time.

The state synchronizing method based on CRDT works by storing state in a data structure that is of the conflict free replicated data type. The benefits this data structure provides lie in merging multiple different states. Whenever client changes their state and wants to synchronize his new state with all other clients, all that is required is to send a change operation.

The simplest examples of such conflict free data structure are a counter and a set, where both have only *get*, *add* and *subtract*, or *remove* operations allowed. We can see that no matter what operation is received, it's impossible to arrive at a conflicting state. Every time there is only one way to merge the operation, with deterministic result.

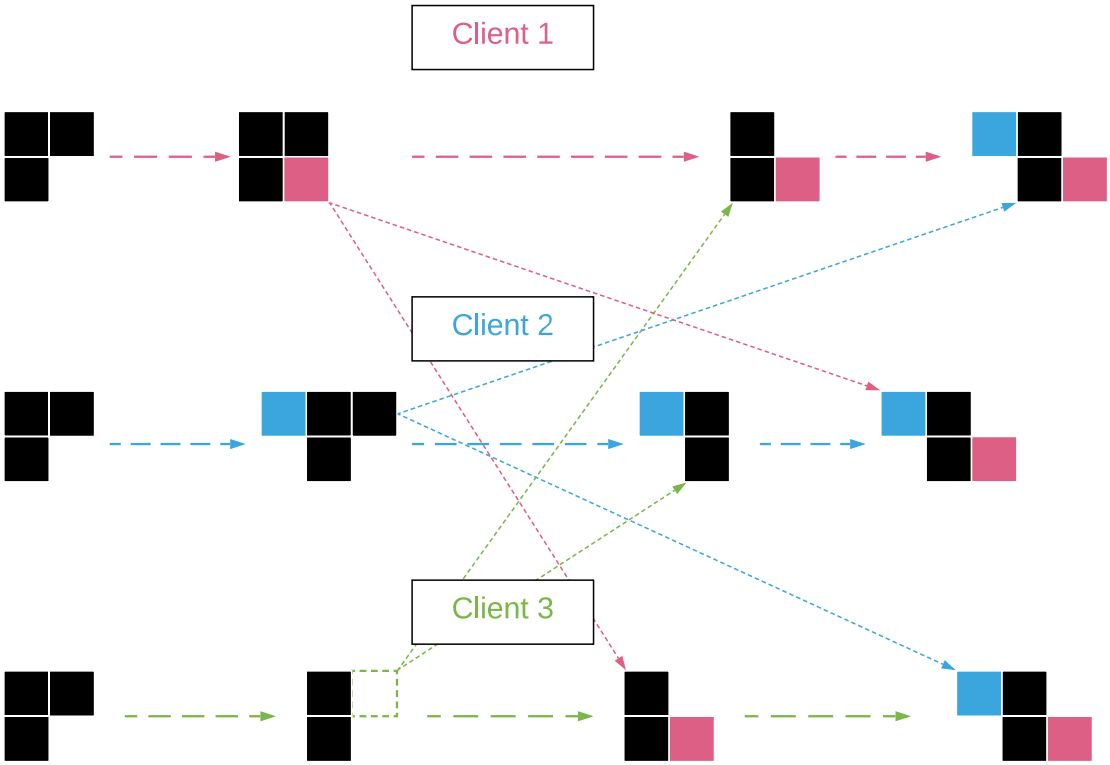

Figure 4 CRDT set synchronization

This is another point that makes CRDT attractive to developers. It makes merging state easy and intuitive as it guarantees that no state conflicts will ever appear. While simplicity has many advantages, mainly during eliminating edge cases and proving correctness, inability to modify merging behavior can also be considered a disadvantage. Specifically, in our scenario, we would like to ensure no data is lost upon reconnection. Which would be impossible when using CRDT.

3.2.2.3 Operational Transformation

The most academically studied synchronization method is Operational Transformation (also referred to as OT). Described in the late eighties (Concurrency Control in Groupware Systems, 1989), this method is now the most versatile and supported method for state synchronization.

The idea of OT synchronization is based on finding the intent of each submitted operation performed on the document and transforming it in a way, that the original intent of the change is preserved. This means, that if one client deleted the last word of the document, while other client moved the last word to the front, the resulting intent would be to delete the word in the front.

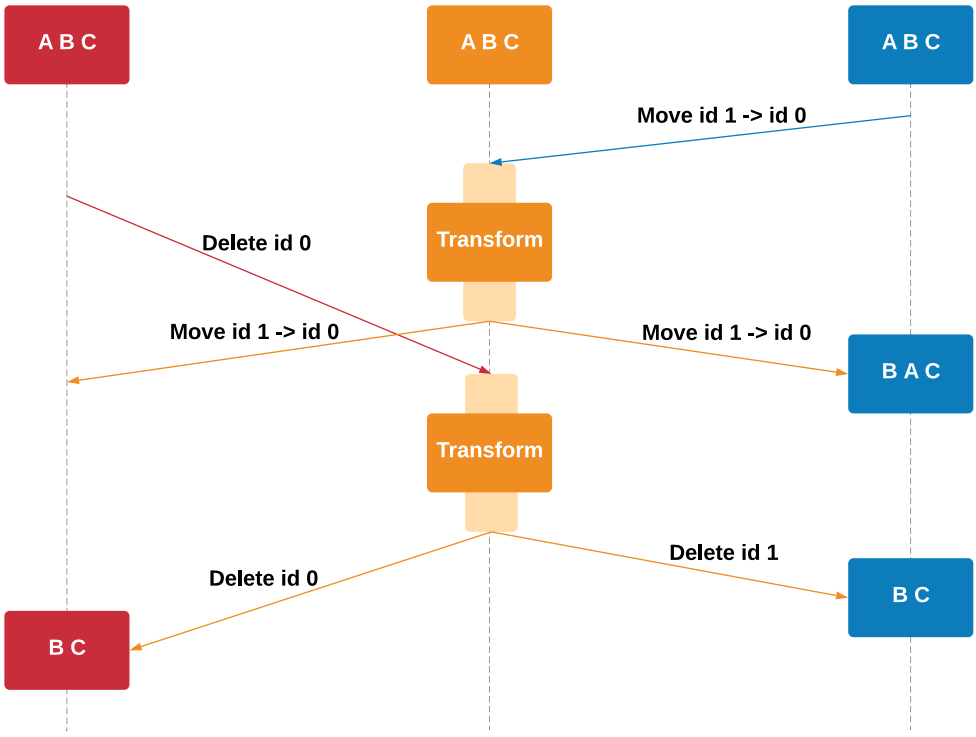

Figure 5 Operational Transformation

Preserving the intent of changes increases the chances that the changes performed while offline will be merged correctly[6] on reconnection. However, this preservation gives us no guarantee that the intent is guessed correctly and that it won't change in case the document state changes. For this reason, we cannot rely purely on the built-in merging mechanism in our offline mode.

Besides working in a Client-Server architecture, ideal for offline mode implementation, the main reason to consider OT as a synchronization method for us is its versatility. Various extensions of this method can be used to solve problems with offline mode implementation:

- Traditional OT supports transforming received operations only in one way, such that clients are only able to add to the history of performed operations. Conveniently, it is possible to implement a reverse transform function such that using history of stored operations and current state it can perform undo operations and create a view of the edited document displaying its past state. This is extremely useful when merging state after reconnection, because anything that was already stored on the server is guaranteed to be kept, preventing any possible loss of already stored data.
- Another extension of this synchronization method improves its space efficiency and ability to perform fast reconnections. This is achieved by merging operations performed while offline into a fewer, more complex and compact operations.

However, this method is not without problems. The first implementation challenge is storing client state for offline mode. Saving all the required data and flags without creating invalid state is very tricky, as the implementations of OT relies on several arrays with operations that may not be valid after reopening new browser session. Example of such array is list of inflight requests that are roundtripping to the server.

Another issue appears once we start synchronizing state after reconnecting. Due to the nature of this method, there is no simple way to show the user full difference

[6]Merged in a way that results in an equal state to the state we would get by performing a manual merge.

between server state and client state. The only available alternative is viewing the difference one operation at a time, which may have affected different parts of the document. Viewing the difference in this way would lead to user having to scroll up and down the document several times, just to perform the merge, before the synchronization is done.

3.2.2.4 Differential Synchronization

A lesser known synchronization method, developed by Neil Fraser from Google is Differential Synchronization (Neil Fraser Google, 2009). This method takes inspiration from the idea employed in version control systems. These systems don't require constant network connection, making DS a strong candidate for our offline mode implementation.

The main benefit of this method lies in the fact, that changes don't have to be tracked as they are performed. This means we don't need to track each operation performed on the edited document. As a side effect it also assures us that time or space complexity will not increase while working offline, because synchronization request payload calculation is constrained by the document size, rather than the number changes.

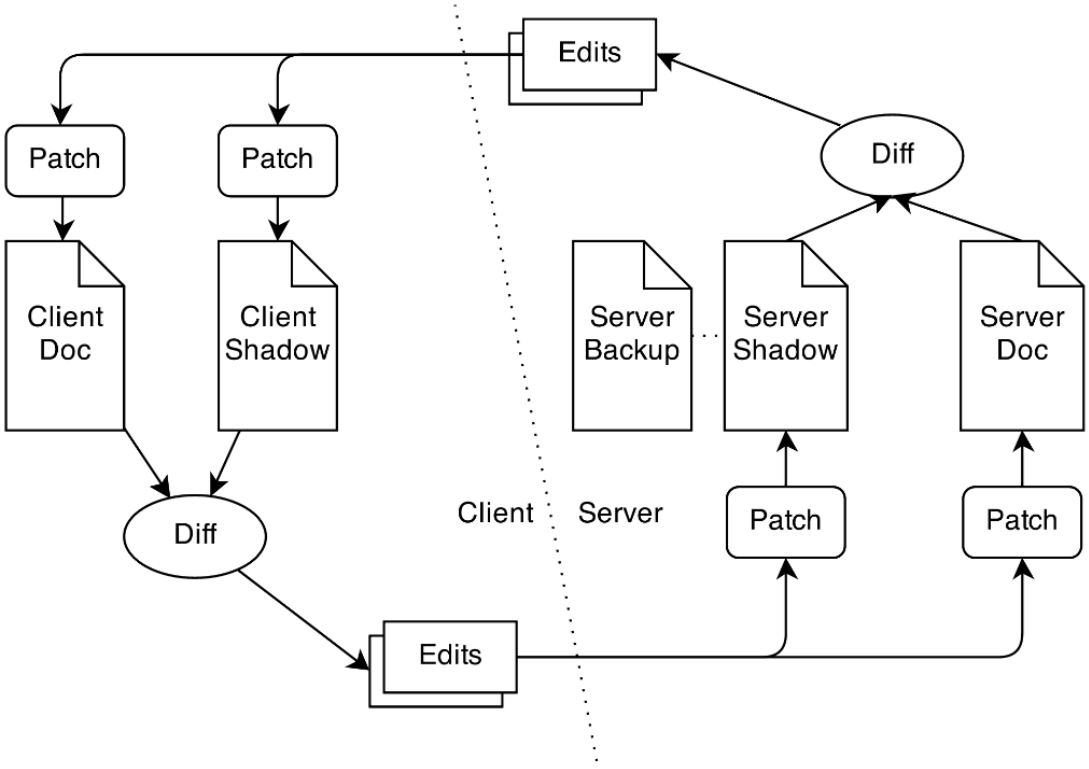

Figure 6 Differential Synchronization Architecture

Image source: https://neil.fraser.name/writing/sync/

Since differences are calculated, rather than stored, implementing offline storage will be a simple task without any edge cases, where only a copy of the current state is stored. This difference calculation, however, needs two versions of the state to work. One with client changes and the other, serving as a state copy shared by the client and the server, this copy is referred to as shadow in the papers by Neil Fraser. In case of larger state, it is possible that we won't be able to fit both client state and server state copy inside browser storage, posing a serious issue for our offline mode.

Same as with storing data for offline use, performing reconnection merge is as simple as merging the current state with the server one and submitting a final difference. There is an issue with this merging. Since the method does not preserve intent of performed actions, automatic fuzzy merge does a bad job at doing what the user intended to do and so it might delete some of the users work.

To battle the issue of fuzzy merge imperfections, one option is to implement a history of changes that would allow undo and commit attribution, by storing each difference the server receives in a database and implementing a reverse patch function for both the client and server that would take difference and return a state before the difference was applied. The other option, inspired by the Git version control system is to force the user to perform a manual merge and only then submit the changes.

Another, albeit minor issue of this method is that it requires keeping shadow for each client. Which should be garbage collected in the original design as they are no longer valid after the client disconnects. This is not the case with offline mode. The client can disconnect and after reconnecting later the server should be able to recover and keep using the stored shadow.

3.2.3 Optimizing Space and Time Complexity

When working in online mode, performed changes can be submitted periodically and freed from the memory once server acknowledgment arrives. In offline mode however, this is not an option. Until the client regains connection, changes are accumulating, which could result in unacceptable memory consumption. For this reason,

we will define a way to measure space complexity of our solution, which we will take into consideration during implementation.

Various synchronization methods operate on different aspects of state or operations. Each of these aspects pops up in some way in memory or time complexity of their synchronization. We will refer to the state size itself as a n, history of performed operations as h, and the number of unsubmitted operations as c.

Another issue is a time complexity of merging changes. A $O(n^2)$ merge algorithm is acceptable, when there is only one change submitted at a time. However, in case of reconnection the client might suddenly submit too many operations for server to handle. This would result in a situation almost equal to not sending any data at all.

The reason space complexity is important to us are also browser storage limitations. Depending on the browser and storage mechanism, tight memory limits as low as 2.5MB are set up for each application. If we don't store the state or operations in an efficient manner, we might run out of space and start losing client data on page refresh.

3.2.4 Resolving Significant State Conflicts

The biggest challenge offline mode presents is the way merging state conflicts that could lead to loss of important data is performed. Since client can be offline for a longer period and by the time of reconnection, the client and server state can differ widely, and automatic conflict resolution behavior might not be the best solution as it could silently delete important parts of the server state or client changes.

Since automatic merge is out of the question our options for resolving these conflicts are very limited. For each operation, difference or changed part of the document we must ask the user how the conflict should be handled.

This prompting the user can be done in several ways. One simple, although not user-friendly way is to show a dialog for each operation received, asking the user if the operation should, or should not be applied. This solution not only lacking in user

experience, it also does not allow creating a state that is mix[7] of both server and client state.

A very robust solution is showing the user preview of both server and client state and asking him to create a new state with both previewed states manually merged into it. This way, also called two-way merge we can ensure that merging is done correctly and without much impact on user experience if the merging is not required often.

The last issue with merging state conflicts is with states stored in data structures different from strings and arrays. For merging conflicts in maps, or JSON objects, a system must be defined, for translating between manually merged changes in our data representation into a JSON representation and back.

3.2.4.1 Detecting Large State Changes

Before we start the manual merge process, we also need to ensure that it is needed. In case the user has not performed any changes while offline or was disconnected for a short period of time, which happens if the client has unstable internet connection, manual merge might not be required, and we can skip the step entirely.

Asking user to resolve state conflicts too often would decrease user experience of applications using the offline mode. For this reason, asking for manual merge on every reconnection is unwanted behavior. Which leaves us with question: What metric do we use for triggering the manual merge procedure?

In all cases we want to prompt the user to merge changes only after the reconnection happens. One metric we can introduce is how much different is the server state from the client state, either by counting the operations or by looking at the size of received state difference. The drawback of this method is that it has no way to tell how important the made changes are. Even when not much has changed, the introduced changes might be too significant to leave them to a fuzzy merge mechanism.

[7] If the original state was "A", client one "**A**" and server one "AB", by mixing the states during merge, we could arrive at "A**B**" state, that combines both server and client one. This is achieved by manual merge without directly accepting, nor rejecting the client change, which would result in either "AB" or "**A**B" state.

A feature that solves this problem is using time passed from last server synchronization as a metric. This way we would perform fuzzy merge only in cases when users see the results and are able to correct them in case of incorrectly merged state. This could potentially prompt the user to perform a manual merge over a singled fixed typo in text, just because he was offline for longer time.

Thankfully both approaches can be combined to build a solution that would minimize the drawbacks of each and prompt the user only in cases when manual merge is the only way to guarantee good user experience.

4 Defined Architecture

Now that we have each subproblem defined, all that is needed for a working offline mode solution is to define a viable solution to each subproblem. We will proceed with suggested architecture for offline mode that fulfills all requirements we set in the second chapter.

This architecture is kept abstract and free of implementation details so that you can implement it freely using any technology stack. While this architecture only serves as a reference, we recommend following it as decisions made in resolving one subproblem often affect the solutions of next subproblems offline mode implementation presents.

4.1 Choosing and Integrating a Synchronization Method

In order to implement an offline mode in a collaboration system, we first need a working version of such system working when all clients are constantly online. For this purpose, we need to choose the synchronization method most suitable for our requirements. Once this method is chosen an offline mode can be written on top of it an extension.

We ruled out CRDT (see chapter 3.2.2) since its peer to peer architecture does not allow us to easily store and propagate changes. These functionalities are present in Client-Server architecture utilized by DS and OT synchronization methods.

Out of these two options we recommend choosing DS over OT as it allows for simple manual merge implementation used after reconnection. This functionality is critical for offline mode implementation and as such should be our priority. Besides this advantage, DS representation of state is much simpler, thus removing many edge cases when saving and restoring state in browser.

4.1.1 Server-side integration

Since DS operates using `diff()` and `patch()` functions, it is essential we define the format in which differences are sent to the client[8]. We do this in a way that client can

[8] The other way around, from the client to the server can remain unchanged, as the server never performs manual merge.

easily display information needed for a manual merge procedure. We recommend sending differences with following information:

- Operation result, stating if the following content was added or removed[9].
- Position of the change. This could be an XPath or line number depending on a format of our state.
- Content that should be removed or added inside patch algorithm.

Another major change that needs to be introduced on the server is a concept of endless sessions. Since client now keeps his data stored indefinitely and can reconnect at any time, it is needed to persist the shared shadow on the server as well. This can be done by treating each session as a cached object that is primarily stored in the backend database. If memory starts running low, or some time passes since the last request from the client, last used shadows are persisted and forgotten to free up server resources.

An optional improvement upon the concept of endless sessions is a disconnect request. Before the client closes his application, he can choose to manually help server free up his session by sending a request indicating that no additional communication will occur in the current session.

4.1.1.1 Persistence

By introducing the concept of endless sessions, we quietly introduced a new requirement for our Server-side architecture. Since losing common shadow on the server would result in a situation, where server is unable to perform patch on client data, we need a way to persist this part of session data in all cases[10].

The general approach for persisting data, introducing database to the system, is enough here. The database can store client shadow when session is cached out. And later, queried back using unique user, or session identifier for later use.

[9] This result is usually displayed as a + for added and – for removed content on each line the change relates to.
[10] Together with shadow a backup shadow should also be stored. This backup is also documented in the original DS papers and is used to ensure consistency between the server and client even in edge case scenarios.

In an extended version of the system, this database could also be used to provide version history, for undo support. In this case, each difference would be stored together with its session, or user id and some version identifier, in a form of timestamp, or incremental counter.

4.1.2 Client-side integration

With all the server-side requirements out of the way, we can move onto the client part. As client is the main consumer of our offline mode, the architecture on this end requires more changes than on the server.

4.1.2.1 Reconnecting from Offline Mode

The first and primary requirement for offline mode is ability to reconnect. For this purpose, a simple *ping* request can be introduced (See chapter 3.2.1). In case any of our sent requests does not receive a response from the server we can declare that we are in a disconnected state and setup our reconnection discovery mechanism. This works by periodically, in equally long, or increasing intervals, sending the *ping* request to the server and waiting for response. During this pinging we assume no response is returned. In case when server responds we are reconnected and can start merging our state with the server one.

As a part of server *ping* response, the difference between server state and client shadow should be sent. Using this difference, we decide if manual merge is needed and ask client or automatically merge according to that. After merging is successful, we now have all the server changes locally, however, server still does not have changes from the client. To fix this, the client can now freely calculate difference and submit the changes to the server just like in the online mode.

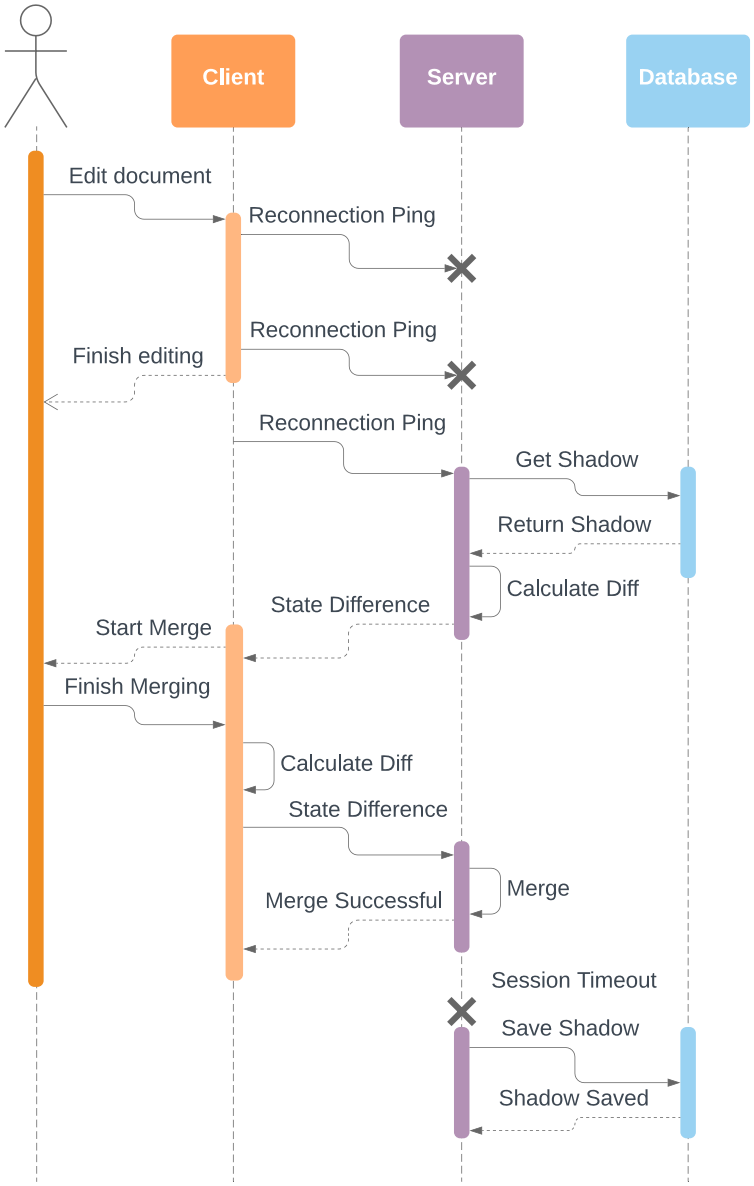

Figure 7 Offline Mode Reconnection

Cases where connection gets lost after the client has merged his state and updated shadow, but the changes got lost on the way to the server, or where packets sent from the server don't arrive to the client are handled just like in online mode and don't require any additional changes.

4.1.2.2 Persisting Local Changes

Another architectural change that is required on the client is keeping all the performed changes even after the browser session ends[11]. First, this ensures that no data is lost after refreshing the webpage, when users start making sure that connection is in fact lost. Just like with flipping light switch, when electricity is out. It is natural for users to attempt to reload websites when they stop responding.

The secondary purpose of this persistence is allowing users to work even when no connection is present. This way they can freely open the application, make some changes and close it without the fear of losing any work they had just done.

When offline mode is activated, instead of attempting to submit the changes to the server, new snapshot of client state is saved. This snapshot is later retrieved when reloading the application and cleared every time changes are successfully submitted to the server[12].

4.1.2.3 What Data Needs to be Persisted?

We have mentioned persisting changes, however, depending on the synchronization method of choice there is no single variable holding all the changes. Differential synchronization operates on calculated differences[13], client state and shadow which should be same on both client and server and a session identifier used for endless sessions.

Even though differences can be calculated each time without any significant performance, there is a need to persist them. In edge cases where packets are lost on the way between client and the server, they are sent together with their version identifier in order to synchronize shadows back to a consistent state. The other attributes of our system, however, need to be stored. Shadow and session id, mapping server shadow to this client session need to be saved in order to allow patching and calculating differences

[11] This happens when browser tab is closed. Reloading or refreshing the website does not end the session.
[12] If your implementation of significant state conflict does not trigger manual merge if no client changes were performed, this step is optional as upon reloading, synchronization with the server is required anyway.
[13] Which are stored if they were scheduled for submit, but server response from the last change submit hasn't arrived yet.

that are consistent with the server. Client state holds all the performed changes and as such it is important that we also persist this attribute.

4.1.3 Conflict Resolution Changes

We've mentioned previously that DS synchronizes server and client states by performing a patch mechanism. In online mode this is represented by a function that automatically performs a fuzzy merge by taking current state and received differences and returning a new state that is equal to the sender's state at the time of submitting differences.

This behavior needs to be changed after reconnecting as automatic merge might remove important data that isn't easily replaced (see chapter 3.2.4). More precisely it needs to be changed in a way that when significant changes were introduced, we leave the merging on the user, instead of an algorithm.

This reconnection merge happens in three steps. First a difference is received from the server. In the second step a significant state change detection algorithm gives us the information if manual merge is in fact required. It it's not, we proceed by doing the regular automatic fuzzy merge. Otherwise a new state with current state and server differences is displayed in a user-friendly manner for merging. External diff viewers[14] can be used for this purpose.

While user performs the manual merge, we can detect that all conflicts were resolved. To further increase the user experience of applications using our offline mode and prevent any parsing errors[15], state is periodically checked if all conflicts were resolved and document state is valid, instead of giving the user a submit button for confirming that he has resolved all conflicts.

After such significant conflict is resolved, merged state can be submitted to the server and offline mode disabled by clearing persisted data and resetting any variables indicating that offline mode is enabled.

[14] Program that lets user resolve merge conflicts inside Git version control system.
[15] Information added to the state for the purpose of merging could cause issues if not removed.

4.1.4 Detecting Conflicts that Require Manual Merge

A significant state conflict detection algorithm must be present for the offline mode conflict resolution to work. The ability to tell when significant state change occurred is used to prevent frequent manual merges for users that have a working, although frequently outing internet connection.

There are several ways to do this detection (see chapter 3.2.4.1), all of which are heuristic in nature. Out of the possible options we concluded that a solution incorporating both, time tracking, and difference size would yield best results.

As a good starting point, we declare any state difference as significant if both options are true. If more than 2 minutes[16] have passed since the last successful submit[17]. Additionally, if the changes client performed any difference in the time that has passed since last difference calculation. If no changes were made on the client side, we can quickly accept any server-side changes without the fear of incorrect merge.

[16] This value can be tweaked depending on the network conditions and severity of manual merges.
[17] Tracking time from start of the offline mode could be unreliable as it is started only after we discover network partition. This discovery could potentially occur significantly later than the partition occurred. Therefore, it's required to track time from the last success response from the server.

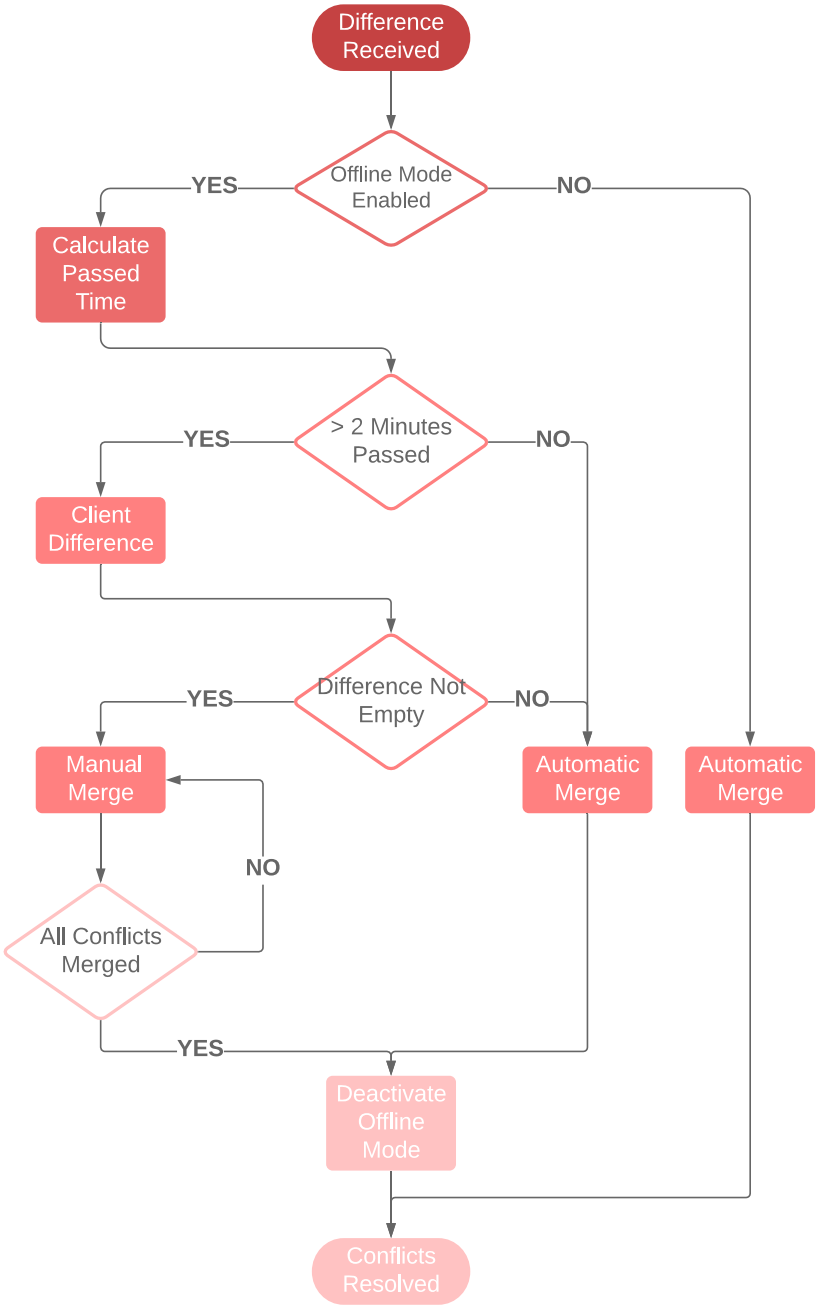

Figure 8 Conflict Resolution Diagram

In the cases when online mode is enabled, or server has sent empty differences, this procedure is skipped as we can guarantee that no significant state changes have occurred.

5 Implementation Details

We started out implementation by looking for already existing implementations and libraries that we could use in our offline mode. This way we could significantly decrease the amount of code we have to write to achieve the same result.

As our architecture relies on already existing solution for online synchronization using DS method, we searched for libraries that could provide this functionality on both the server and the client. One such library that we have found was diffsync (Monschke, 2015). Since our offline mode could not work without this library, we decided to extend it with our solution.

5.1 Library Extension

After a closer look at the library's code, we noticed that significant parts of the DS synchronization method were not implemented. These include using backup, which is crucial in cases, where packets get lost. This backup is only stored, but never used on the server and on the client, it's not stored at all. Acknowledgments for sent requests are also handled incorrectly and instead of removing only acknowledged edits, all edits are removed[18], once a specific type of acknowledgment is received.

Due to these braking changes we decided not to use this library. Rather than fix all found and possibly hidden issues with its implementation, we decided to create our own implementation of DS. Our code is heavily inspired by the diffsync library, however with special attention paid to edge cases and error handling.

5.2 Server Implementation

While server side is free to use any other DS implementation, due to simplicity and consistency reasons we decided to use our new library on both client and server side. To calculate differences and perform patches, the library utilizes jsondiffpatch (Eidelman, 2018) library. Because same library is utilized on both ends for difference processing, there is no need for additional mapping between client-side and server-side generated differences.

[18] Even those that were never sent.

As a first modification, we have changed the format, this jsondiffpatch library uses, when sending differences from the server to the client. In our case, disabling detection of moved object was enough to comply with our defined architecture.

5.2.1 Server Persistence

When it comes to persisting user changes, we had to extend the original library design with two new mechanisms. The first one being endless sessions and the second one, dependent on endless session implementation is client change persistence.

The endless session implementation is primarily server-side. It encompasses a session id generation[19] and adding a persistence mechanism. Since our library already supports database connection using an adapter pattern all that was left to do is extending this adapter interface to support session id.

For simplicity reasons we left out any caching on the server side and all storage is done using supplied persistent data adapter. If needed caching can be implemented as a part of this adapter in a way that's similar to the way Java Persistence API handles data caching.

For the id generation we used a custom UUID4 generator, as this format is widely supported by many databases. If more efficient solution is needed and supported, developers using our extension can easily override our default id generation as a part of their database adapter.

```
function generateSessionId() {
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, c => {
    let r = Math.random() * 16 | 0, v = c == 'x' ? r : (r & 0x3 | 0x8);
    return v.toString(16);
  });
}
```

[19] Id generation is done on the server-side, as opposed to the client-side, because the id must be stored in a database and allows us to use autoincrementing ID, or any other format easily stored in a database like UUID. Additionally, this removes the coupling between client and backend database implementation.

5.3 Client Implementation

On the client we had to implement an offline storage solution, together with reconnection mechanism that can detect that server is reachable again and then merge states without any important data loss.

5.3.1 Reconnection implementation

The first architectural change we made to the diffsync library, on the client was adding a reconnection mechanism. This consists of catching an error when trying to synchronize with the server. An error is guaranteed to be returned only on network errors, when using *fetch* API.

This reconnection mechanism consists of periodically pinging the server and waiting for its difference. For better user experience we progressively increase the time interval that happens between reconnection ping requests by one fifth of the previous time. If the user expects to be offline for half an hour, there is no need to send a ping request every ten seconds.

5.3.2 Large conflict resolution

Once we get the initial response from the server back with data to merge, we start by deciding if automatic merge can be performed. By default, we consider a significant merge anything that happened after thirty seconds after last server connection and contains any edits, since there is no reason to start manual merge in case server state has not changed in the meantime.

After a decision is made, that manual merge must be performed we give the developer using our library the choice of how this situation is handled. A callback with provided merge implementation is called. This callback provides the developer with current state and a merged one and returns a merged state so that he can let user perform the merge in a user-friendly way.

After a manual merge, merge result is sent to the server and offline mode disabled upon successful response. If sending the request fails, we keep sending ping requests, but keep the merged state.

5.3.3 Client-Side persistence

To allow the same kind of flexibility with persistence on the client as on the server side. We decided to also use adapter pattern for the client-side storage of performed changes. The API for this adapter is similar to the one for server-side persistent storage, with `getData()`, `clearData()`, `setData()` and `hasData()` methods.

This adapter is supplied with identifier of shared document[20] and all client-side data related to the synchronization. This includes shared document identifier, endless session identifier, local state, shadow, backup, unconfirmed edits and both client and server document versions.

5.4 Testing

To make sure our solution works we tested the implementation as it was created. Most of the testing was done manually by performing some changes on multiple opened browser windows and disconnecting from the network on a demo GUI project, using a rich-text editor. We observed both simple scenarios, where no conflicts were present and scenarios where clients directly changed same data.

To improve the stability of our library and to decrease the change of any bugs in our code, we introduced Typescript as a development dependency into our project. This allowed us to use types and enforced many rules, which were able to detect hard to find errors in our code.

Additionally, to make our implementation as stable as possible, together with the library we provided some unit tests that would detect any breaking changes upon our offline mode implementation.

5.4.1 Reasoning Behind Stability of Offline Mode

We have based our implementation on papers by Neil Fraser on Differential Synchronization. There he goes into detail about why his solution works, arrives at a consistent state and never leads to a data loss.

[20] In the code we refer to this shared document as room, to keep similarity with diffsync library. Its terminology was based on the terminology of socket.io, where rooms are used for broadcasting.

If our offline implementation does not introduce any new concept to the online solution, it is also guaranteed to work with already mentioned characteristics. Therefore, we made sure not to modify the way the method works in online mode and only added a reconnection mechanism based on already proven principles.

In cases, where all communication is done in loops, where one of the sides initiates a synchronization request by sending all edits and the other side responds with its edits, the online mode is guaranteed to work. This holds true even for cases, where either packets, whole request, or response are lost.

In the case of reconnection, we create two such loops. The ping request stimulates the start of the first loop, where server response to the ping is equal to a synchronization request starting the loop. Next after merging, the state is submitted to the server, which, while is sent as a request, serves as a response to the first sync loop.

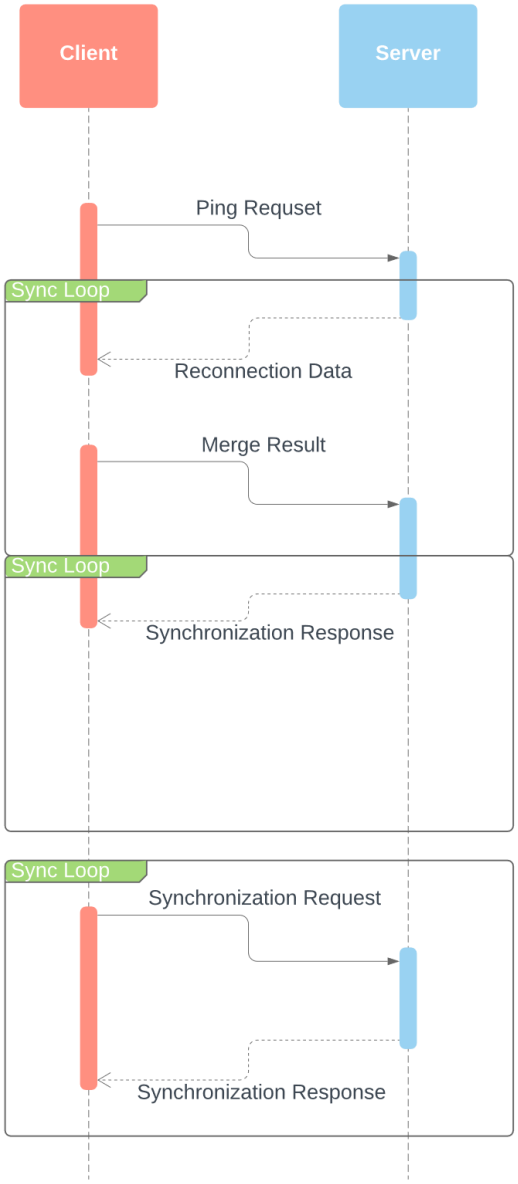

Figure 9 Synchronization Loops on Reconnection

The actual response sent from the server now serves as a start of the second loop. This loop simulates a behavior of packets lost on the way back to the initiator, by not sending any response. However, this is fine, as on the next request sent, backup is used to rollback on the next incoming sync request, returning us to a valid state.

6 Solution Analysis

With our solution designed and implemented it is now time to reflect on its characteristics. We will start with the biggest benefit and that is time complexity. Since our solution does not accumulate changes like other synchronization methods, its time complexity is purely based on the size of state.

Calculating unsubmitted differences has the same characteristics as a Longest Common Subsequence algorithm. This problem is solved using dynamic programming in time $O(n^2)$ with the space complexity being $O(n)$ after some optimizations. Merging states is significantly faster with only $O(n + d)$ time and space complexity, where $n + m$ becomes the new state size.

The problematic part are client persistence limitations. While we solved this issue by introducing adapter pattern and let another developer decide how the data is stored, currently there is no solution that guarantees ability to store large state, also with its backup and shadow. Minor improvement like running a compression algorithm before storing our state would help with this problem, while also slow down the saving process.

While offline mode is working, users can't undo their operations, or preview change history. In case some user makes mistakes during manual merge, this change will be definite after submission with no way of returning to the previous state. In the future we could enhance this mode by adding a reverse patch function that would take difference stored on the server and revert the state to its previous form. Such improvement would also allow commit attribution, where we could map each section of the document to the user that added it.

Conclusion

As a result of this thesis we implemented an offline mode solution for collaborative systems. This solution fulfills all requirements we set in the first chapter for how an ideal offline solution works.

It allows users to continue working even in unfavorable network conditions. Frequent, or long disconnections don’t cause any collaboration issues when using our solution. With original real-time online solution used when possible with no document locking needed.

We designed the offline mode so that it does not require or use any proprietary or paid solutions. As a part of open source library, any developer designing a new collaboration system, or using a system relying on the same library can make a use of our implementation. Without any impact on how the online mode was previously handled.

Our proposed architecture guarantees that users will lose no work even after they run out of battery, must stop working or simply close the browser by accident. Retrieving previous data and reconnecting presents no issues for users with unstable internet connection no matter if it just goes out for short, long time periods.

Resume

V tejto práci sa zaoberáme riešením problémov spôsobených odpojením klienta zo siete, ktorá synchronizuje stav medzi klientami. Tento problém sme sa rozhodli riešiť, pretože v mnohých sférach prace, napríklad medicínskych, ci komerčných sa používajú kolaboračné systémy, ktoré vyžadujú nepretržité pripojenie k sieti.

Táto požiadavka spôsobuje, že mnoho ľudí nie je schopná pracovať, alebo priamo stratí všetky zmeny, ktoré sa nestihli poslať na server pred odpojením, pričom nespoľahlivé siete sú faktom života. V lepšom prípade takéto odpojenie spojene so stratou dát len spôsobí frustráciu, v tom horšom to môže ohroziť ľudské životy, ak sa diagnóza stratí počas synchronizácie so serverom.

Preto chceme navrhnúť offline mód, ktorý by bol jednoducho implementovateľný. Nevyžadoval by žiadne neprístupne, ci platené riešenia a taktiež by neafektoval správanie fungujúceho online riešenia. Tento mód by dovolil používateľom naďalej pokračovať v práci aj v prípade že nemajú aktuálne spojenie so serverom. Garantoval by im, že ich dáta sa nestratia ani po uzavretí webového prehliadača a taktiež, že v prípade, keď bude synchronizácia znovu možná, sa nestratia žiadne doležíte dáta.

Našou snahou pri tomto móde je taktiež zlepšiť zážitok používateľa pri práci s kolaboracnými systémami a teda sme brali ohľad aj na rýchlosť a dostupnosť nášho riešenia, tak ako aj nutnosť používateľa manuálne rozhodovať ako sa majú zmeny zo servera premietnuť do lokálneho dokumentu. Primárne sme na tento fakt prihliadali pri znovu pripojení, kde sa naďalej vykonáva tichá a automatická synchronizácia v prípade, že vieme jednoznačne spraviť tieto rozhodnutia.

Pred tým, než sme náš problém začali riešiť sme analyzovali aktuálne dostupne kolaboračné systémy a rôzne riešenia offline módov. Týmito riešeniami sme sa neskôr inšpirovali a používali ich kladne vlastnosti v nami navrhnutom systéme.

Prvým populárnym kolaboračným systémom, ktorý sme analyzovali bol Office 365 od spoločnosti Microsoft. Tento systém plne zlyhal pri strate pripojenia. Zatiaľ čo

používateľovi zobrazil hlásku, že sa jeho zmeny ukladajú, reálne k tomuto uloženiu nedošlo ani po znovu pripojení do siete.

Zaujímavo tento problém riešil kolaboračný systém Box, ktorý v prípade, že sa používateľ odpojil mu systém naďalej dovolil pracovať. Následne, aby sa vyhol synchronizácii a potenciálnej strate dát, uložil dokument ktorý vznikol počas offline režimu ako kópiu zdieľaného s číslom na konci. Toto riešenie avšak nedovoľuje používateľom pohodlne spolupracovať na jednom dokumente a vyžaduje, aby opakovane manuálne tieto dokumenty spájali do jedného.

Spoľahlivým riešením kolaborácie s offline módom je Git verzovací systém. Klienti v tomto systéme fungujú primárne offline a len v prípade, že sa rozhodnú svoje zmeny zdieľať si vypočítajú čo sa zmenilo, pripoja sa na server a zmeny odošlú. Aj keď z dôvodu chýbajúcej spätnej odozvy je toto riešenie pre naše účely nepoužiteľne a závažne by zasiahlo do online módu, koncept, akým sa posielajú zmeny len v prípade potreby a ako sa garantuje že sa dáta nestratia vyhovuje mnohým našim požiadavkám.

S predstavou, ako fungujú aktuálne existujúce systémy sme si zadefinovali náš problém a rozdelili ho do menších pod problémov, z ktorých sa skladá. Ak budeme vedieť vyriešiť každý jeden z týchto pod problémov, budeme mat fungujúce offline riešenie.

Prvým zadefinovaným pod problémom je detekcia znovu pripojenia. Webové prehliadače primárne fungujú s internetovým pripojením a teda aj metódy, ktoré ponúkajú na detekciu toho, že sa pripojenie na server stratilo a následne znovu sprístupnilo sú veľmi limitovane a neriešia mnoho krajných prípadov.

Jediným spoľahlivým riešením je opakované posielanie požiadaviek na server s očakávaním, že zlyhajú. V prípade že nejaká z požiadaviek vráti validnú odpoveď, vieme, že server je znovu dostupný zo siete a teda môžeme začať synchronizáciu.

Ak chceme riešiť konflikty, ktoré vznikli po dlhšom čase na klientovi, musíme dobre poznať synchronizačnú metódu, ktorá sa používa na dosiahnutie konzistentného stavu počas online režimu.

Najjednoduchšou z týchto metód je zamykanie dokumentu. Napriek jej intuitívnosti a jednoduchosti, toto riešenie zlyháva, keď klient stratí pripojenie a neuvoľní zámok nad dokumentom, alebo jeho častou. Dodatočne, aj keď zamykanie predíde konfliktom v stave, nevieme ho využiť počas offline režimu.

CRDT a OT sú aktuálne najpopulárnejšie metódy na synchronizáciu, kde obe fungujú na baze propagovania vykonaných operácii.

V prípade Peer-to-Peer CRDT ide o posielanie vykonaných operácii medzi všetkých peerov v sieti a dátová štruktúra, nad ktorou sa tieto operácie vykonajú garantuje, že sa klienti dostanú do konzistentného stavu.

OT rieši problém synchronizácie tak, že centrálny server ma prehlaď o stavoch jeho klientov a pri prijatí nejakej zmeny ju pre každého klienta transformuje tak, aby sa po jej aplikovaní klient priblížil ku konzistentnému stavu.

Tieto metódy sme však vylúčili pri našom offline riešení kvôli ich Peer-to-Peer architektúre v prípade CRDT, či kvôli komplexnosti a problematickosti implementácie manuálneho spájania stavu.

Poslednou metódou je DS, ktorá sa inspiruje verzovacím systémom git. Taktiež kalkuluje vykonané zmeny len v prípade potreby, zatiaľ čo je druha strana schopná tieto zmeny dolepiť do aktuálneho stavu. Zo strany implementácie offline režimu, táto metóda má nedostatky len v kalkulácii zmien, kde už nie je jasné, akým spôsobom tieto zmeny nastali a teda automatické spájanie stavu častejšie vygeneruje nesprávny stav.

Keď budeme mat zvolenú synchronizačnú metódu, ktorá je schopná offline režimu a taktiež efektívnej synchronizácie, počas ktorej používateľ nebude musieť čakať

na server, kým pošle všetky zmeny, musíme určiť, ako sa budú riešiť stavové konflikty počas znovu pripojenia.

Je zjavné, že pokiaľ sa počas online režimu istá časť stavu nesprávne spojí do finálneho stavu, používatelia, ktorí konflikt spôsobili budú schopní okamžite túto chybu napraviť. V prípade offline módu je avšak potrebná špeciálna pozornosť používateľa na to, aké zmeny prebehli, kým bol offline.

Funkčným, aj keď nepríjemným riešením problému je zakomponovanie histórie. V tomto prípade aj keď dôjde k nesprávne spojenému stavu budú používali schopní obnoviť predošlý.

Priatelenejším riešením je manuálne spájanie, kde používateľa požiadame, aby namiesto systému spojil stav on. V tomto prípade je avšak nutné sa uistiť že automatické spájanie nie je vhodné, aby sme používateľa príliš neotravovali. Taktiež je nutné posielať zmeny vo formáte, v ktorom ich bude systém schopní interpretovať používateľovi.

Výsledkom tejto našej analýzy pod problémov je jednoduchý návrh architektúry. Kde sme rozhodli, že na offline mód je ideálna metóda diferenčnej synchronizácie, kvôli jej časovej zložitosti, jednoduchosti implementácie a schopnosti jednoducho integrovať manuálne riešenie konfliktov.

Výsledné riešenie zahŕňa potrebu upraviť serverovú časť synchronizácie s nekonečnými sešnami. Keďže je nutné, aby si server vždycky pamätal, v akom stave sa nachádzal klient v čase poslednej synchronizácie.

Na klientskej strane zase popisuje, úpravy potrebné na perzistentné ukladanie stavu, aby bol používateľ schopní zatvoriť prehliadač bez strachu, že príde o svoju prácu. Taktiež je popísane, ako sa riešia konflikty stavu po znovu pripojení. Ako by mal vyzerať algoritmus rozhodujúci, ci je manuálna synchronizácia nutná a spôsob, akým bude prebiehať.

Túto architektúru sme následne implementovali za využitia knižnice s otvoreným zdrojovým kódom, umožňujúcej synchronizovať pomocou diferenčnej synchronizácie. Po dodatočnej analýze sme našli nedostatky v tejto knižnici a teda namiesto jej rozšírenia sme vytvorili vlastnú, inšpirovanú jej ideami. Naše riešenie je, rovnako, ako táto knižnica voľne dostupné s otvoreným zdrojovým kódom.

V rámci implementácie sme sa zaoberali spôsobom, akým dovolíme čo najvoľnejšiu a najjednoduchšiu integráciu nášho riešenia do existujúcich, či nových systémov.

Naše riešenie sme nakoniec testovali manuálne aj pomocou jednotkových testov. Výsledne riešenie ma akceptovateľnú časovú zložitosť, pričom jeho najväčším problémom je ukladanie stavu klienta, kde môže dôjsť k prípadu, že úložisko nebude schopné si všetky dáta zapamätať.

References

Concurrency Control in Groupware Systems. **Clarence , Arthur Ellis and Simon , John D Gibbs. 1989.** Portland, Oregon, USA : SIGMOD '89 Proceedings of the 1989 ACM SIGMOD international conference on Management of data, 1989. ISBN:0-89791-317-5.

Džama, Jozef. 2017. NoSQL databázy a podpora offline režimu. *Thesis.* Košice : Univerzita P.J. Šafárika v Košiciach PF UPJŠ ÚINF, 2017.

Eidelman, Benjamín. 2018. github.com/benjamine/jsondiffpatch. *github.com/benjamine/jsondiffpatch.* [Online] Jun 25, 2018. [Cited: April 11, 2019.] github.com/benjamine/jsondiffpatch.

Hickson, Ian. 2016. Web Storage (Second Edition). *www.w3.org.* [Online] World Wide Web Consortium, April 19, 2016. [Cited: April 11, 2019.] https://www.w3.org/TR/2016/REC-webstorage-20160419/#the-localstorage-attribute.

Korłub, Waldemar. 2017. enauczanie. *enauczanie.pg.edu.pl.* [Online] November 6, 2017. [Cited: April 11, 2019.] https://enauczanie.pg.edu.pl/moodle/pluginfile.php/253757/mod_resource/content/0/07%20Data%20synchonization.pdf.

Monschke, Jan. 2015. github.com/janmonschke/diffsync. *github.com/janmonschke/diffsync.* [Online] MAY 27, 2015. [Cited: April 11, 2019.] github.com/janmonschke/diffsync.

Neil Fraser Google. 2009. Differential Synchronization. 1600 Amphitheatre Parkway Mountain View, CA, 94043 : s.n., 2009.

Shapiro, Marc, et al. 2011. Conflict-Free Replicated Data Types. [book auth.] Défago Xavier, Petit Franck and Villain Vincent. *Stabilization, safety, and security of distributed systems. 13th international symposium, SSS 2011, , . Proceedings (pp.386-400).* Grenoble : 13th International Symposium, 2011.

Smith, Nate. 2016. https://news.ycombinator.com/item?id=12303100. *Hacker News.* [Online] August 17, 2016. [Cited: April 21, 2019.] https://news.ycombinator.com/item?id=12303100.

Table of Figures

7 Attachments

Attachment A: DVD medium – Our open source project inspired by the diffsync library, containing Differential Synchronization and offline mode implementation.