UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Developing a Local-First Application with Automerge

Stian Økland

INF-3981 Master's Thesis in Computer Science - June 2021

“There is no cloud, it’s just someone else’s computer”
–Twitter sticker

“You have to stop procrastinating that bug fix,
that bug is not going to fix itself ... Probably”
–Delaina Moore

Abstract

In modern time, cloud services have been the go-to approach to store information and documents, and cloud makes documents accessible for users and make it easier to manage items. Cloud is an excellent option to have when data need to be available, but what if collaboration is only needed a few times a week? Once a month? Or unstable connection leading to disconnection to the network? A solution is a Local-first approach. This means all items and documents a user has created is store locally on the device it is created on and provide a feeling of ownership to what is created. Local documents are treated as primary copies, and potential backups stored in an external cloud own by big companies are secondary copies.

Different communication technologies are explored, tested, and evaluated in this thesis in combination with a Local-fist approach and Automerge. A prototype is developed to connect all these ideas and concepts and to better understand which technologies do fit together and provide the necessary features to achieve a Local-first application for collaboration between computers and mobile.

Acknowledgements

I would like to thank my supervisor Weihai Yu for his guidance and help through the process of writing this master thesis.

Thanks to my classmates for coffee breaks, interesting conversations and healthy distraction during the stressful times of writing.

I would also like to thank my family and friends for supporting me through ups and downs during all five years of studying at UiT.

Contents

List of Figures

List of Listings

1

Introduction

Local-first software is a set of principles for software that enables both collaboration and ownership for users [1]. Local-first ideals include the ability to work offline and collaborate across multiple devices while also improving the security, privacy, long-term preservation, and user control of data. Local-first is an "old fashioned" approach of developing a program, often used in programs that require no communication with other programs or systems. In modern time, Local-first has seen more use since offline is not necessarily a wrong state or a system failure in some distributed systems.

Consistency (C) - referring to multiple nodes or sites having the same state of a data set. Availability (A) - refers to data being available at all time. Partition-tolerance (P) - refers to the system being functional and working when the system is distributed over the network, but parts of the system are isolated from each other. One of the challenges, as stated in the CAP theorem [6], is that we can not guarantee strong consistency, instant data update when a site is offline without sacrificing one of the three properties. In combination with Conflict-free Replicated Data Types, the CAP theorem will provide availability, partition tolerance and eventual consistency due to CRDT's abilities when working with replicated data over multiple devices. Consistency will be categorised as strong eventual consistency, referring to devices that will eventually reach the same state when the time taken between updates is of some extend.

CRDTs, or Conflict-free Replicated Data Types, emerged to address the CAP challenges [7]. With CRDT, a site updates its local replica without coordination

with other sites. The states of replicas converge when they have applied the same set of updates. There are many versions and approaches to CRDT, and any of these versions can be combined. State-based and Operation-based are based on how to deploy changes done to a local replica to remote replicas, and versions like add-last-win, last-remove-wins and many more are based on which modification done to the same element win when merged.

1.1 Goals

The main goal of this thesis is to develop a distributed application using Automerge [15] as the core functionality for collaborations. Sub-goals are to experiment, test and decide the additional technology to achieve the main goal. Which type of communication method to use? Which type of devices is used to collaborate through?

The main focus of the application is for users to be able to collaborate with each other, and at the same time, be able to modify local replicas without connection to a network.

1.2 Method

The research method for this thesis is exploratory and experimental, to help find and test technologies to achieve the goal of the thesis.

1.3 Outline

Structure of this master thesis is as the followings.

Chapter 2 - Theoretical background Presents theoretical background about Local-first, communication technologies like Hypertext Transfer Protocol, User Datagram Protocol, and Transmission Control Protocol, CAP-theorem, Consistency, Conflict-free Replicated Data Types, and Automerge. This is followed by a presentation of tools and frameworks containing React and React Native library in JavaScript, as well as JavaScript itself. Next.js and Expo is also presented.

Chapter 3 - Design Presents an overall view of the system, and closer description of design of the server, web application and mobile application. Alternatives

for each design are presented in this section as well.

Chapter 4 - Implementation A section with more detailed explanation of each part of the application, the core functionality of the application and some features.

Chapter 5 - Evaluation Testing methods and testing devices are described in this chapter alongside a discussion of test results and future work.

Chapter 6 - Conclusion A summary and a conclusion of this thesis.

2

Technical Background

This chapter gives a theoretical overview of concepts relevant to developing our Local-first application. Section 2.1 describes Local-first. Section 2.2 will describe Hypertext Transfer Protocol (HTTP), User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) as concepts concerning communication protocols, how they work and what these protocols provide in a distributed application. The following sections are 2.3 CAP theorem, 2.4 consistency in more detail, and 2.5 Conflict-free Replicated Data Types. Section 2.6 describes the cornerstone used in this prototype, Automerge. The last section 2.7 describes other tools and frameworks used to develop this prototype; the JavaScript language, JavaScript libraries React and React Native, the React framework Next.js, and Expo.

2.1 Local-first

In newer time, storing elements in the cloud has been the more attractive option. Storing in the cloud does not only make the creator save disk space on the local device, but all data uploaded to the cloud is accessible from everywhere in the world with an internet connection. However, what if the user needs access to documents or other files where the internet is not accessible? To answer the question, storing elements the "old fashion"-way, locally.

Ownership of notes, documents, drawing, code, or other forms for storable

elements has always been important to the creators of these elements. The first way of storing elements was local, meaning all writes and reads to these elements was done locally, and all other elements on the same local device have access to this element. As mentioned, applications used where the internet is not always available needs a way to store and access elements.

Local-first is undisputed in cases where a user is not sharing any data with anyone and not depending on a connection to get the work done. A Local-first application runs locally on each users device, meaning changes can be done locally without depending on a connection. This achieves complete control for the owner and allows backups, manipulation of files or long-term archiving.

Collaboration depends on the functionality of both Local-first and cloud-based software. Ownership from Local-first combined with cloud as a possible option for backup or storing personal user data. In traditional cloud-based software, instances of a document stored in the cloud are considered the primary copy. All other copies, downloaded or streamed, on local devices are secondary. Local-first application is the polar opposite, where the local copy is primary, and potential backup copies are secondary. The existence of multiple primary copies is possible.

A Local-first approach is not suited for all kinds of application. Application dependent on live updates, communication with other devices in real-time, or latency-sensitive applications are examples of application not suited. Figure 2.1 is a representation of latency around the world. In a Local-first application, latency is not only milliseconds or seconds delay, but also days, weeks or even months. An important factor is that offline is not necessarily considered a wrong state [1].

Another example is store applications. These applications use strong consistency meaning a modification is viewed immediately by all parts of the applications. In stores where money is used to buy items, the transaction has to be atomic, meaning the transaction is fully completed or non of the operations concerning the transaction is completed, resulting in no room for states to be eventually consistent.

2.2 Communication

In a distributed system communication is key to get information and modification across the system. This section describes three different technologies to be associated with the network; Hypertext Transfer Protocol, User Datagram

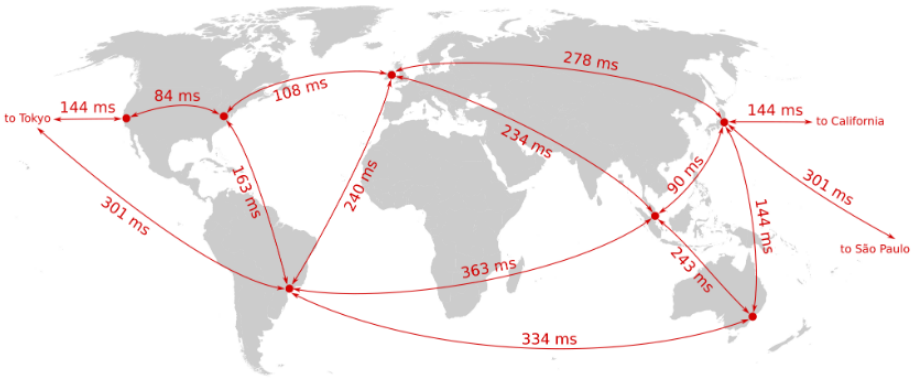


Figure 2.1: Server-to-server times between AWS datacenters around the world. Data from Basilis et al.[2] and figure form Kleppmann et al. [1].

Protocol, and Transmission Control Protocol.

2.2.1 Hypertext Transfer Protocol

Hypertext Transfer Protocol allows fetching resources like HTML, videos, ads or other resources to complete the web page and is the base of data exchanged on the Web. The protocol is a server-client protocol, meaning requests are sent by the recipient (clients), for example, a web browser and the server responds to these requests. A document shown in the web browser consists of a collection of elements from different fetched sub-documents, elements like text, videos, and layout. Figure 2.2 illustrates the workflow of a web document fetching necessary elements from different servers to complete the document.

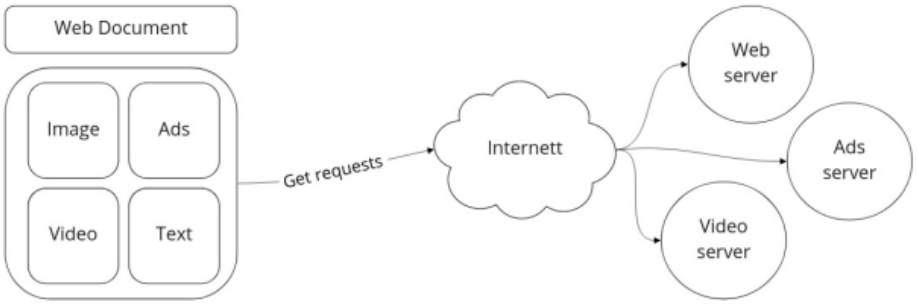


Figure 2.2: Overview of HTTP.

Since the protocol is a client-server protocol, the client and the server is communicating with single messages, request from clients and response from the

server. The most common form of a client is a web browser, and they are always the ones initiating the request. The web browser is requesting an HTML document containing representations of the web page. This HTML file is parsed, then other requests are sent to fetch corresponding scripts, layout, videos or images. The complete page is a hypertext document and can display links to other pages, resulting in fetching another web page based on the link. On the other hand, the server has stored data in different formats and responds to the client requests with the requested HTML document [20].

2.2.2 User Datagram Protocol

User Datagram Protocol (UDP) is a data transport protocol working on top of the Internet Protocol (IP), and in comparison to other protocols, are a simple protocol and are faster over the IP. Since UDP is a fast working protocol, this often results in usage in application with time-sensitive requirements. Such application can be video-streaming applications or online multiplayer games.

User Datagram Protocol is lightweight, meaning mechanism like detecting corrupt data in packages is provided, but detection of out of order packages or missing packages is not.

Each packet sent over IP contains an 8-byte header and variable-length data. Data in a User Datagram Protocol segment is the source and destination port number, segment length and a checksum. A device connected to a network can receive messages on different ports. The difference in port can help to distinguish different types of traffic over the network. Next is the segment length describing the length of the segment and are represented as 16 bits, a 16 character segment of 1´s or 0´s [21].

The checksum is used both by the sender and receiver to check for corrupt data. The sender, based on the data in the segment, computes a checksum and store the sum in the checksum field. On the receiving end, the receiver does the exact computation on the data as the sender and compare it with the senders' checksum stored in the checksum field. If the two checksums are compared equal, there is no corrupt data. If not, some data has become corrupt in the process of sending packages over the network. Listing 2.1 illustrates how checksum works when the sender sends 'Hola', and some data packages get corrupted along the way to the receiver resulting in 'Mola' instead [21].

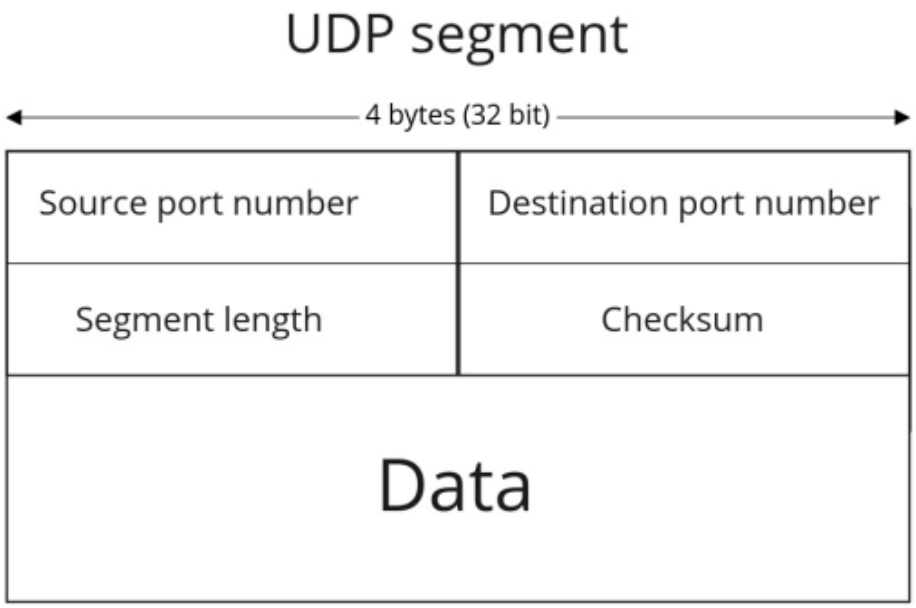


Figure 2.3: Overview of a UDP segment.

```
Hola
01001000 01101111 01101100 01100001
=>  0100100001101111
   +0110110001100001
   =1011010011010000

Mola
01001101 01101111 01101100 01100001
=>  0100110101101111
   +0110110001100001
   =1011100111010000

Differences in checksum
1011010011010000
1011100111010000
```

Listing 2.1: Package corruption example.

2.2.3 Transmission Control Protocol

Transmission Control Protocol is, same as User Datagram Protocol, a transport protocol, and solves many of the problems UDP does not provide a solution

to. These mechanisms handle package loss, corrupt data packages, and out of order packages. Each package sent over the IP contains a header and data, where the header can vary from 20 to 60 bytes. The reason is the options field in the TCP segment [22].

In contrast to User Datagram Protocol, Transmission Control Protocol establishes a connection to the receiving device before sending packages. The protocol is a three-way handshake, where the sending device sends a package with a synchronisation (SYN) bit set to 1. The receiver set its own SYN bit to 1 and responds with an acknowledged back to the initiator. The initiator then sends its acknowledgement back to the receiver to ensure both parts know about each other.

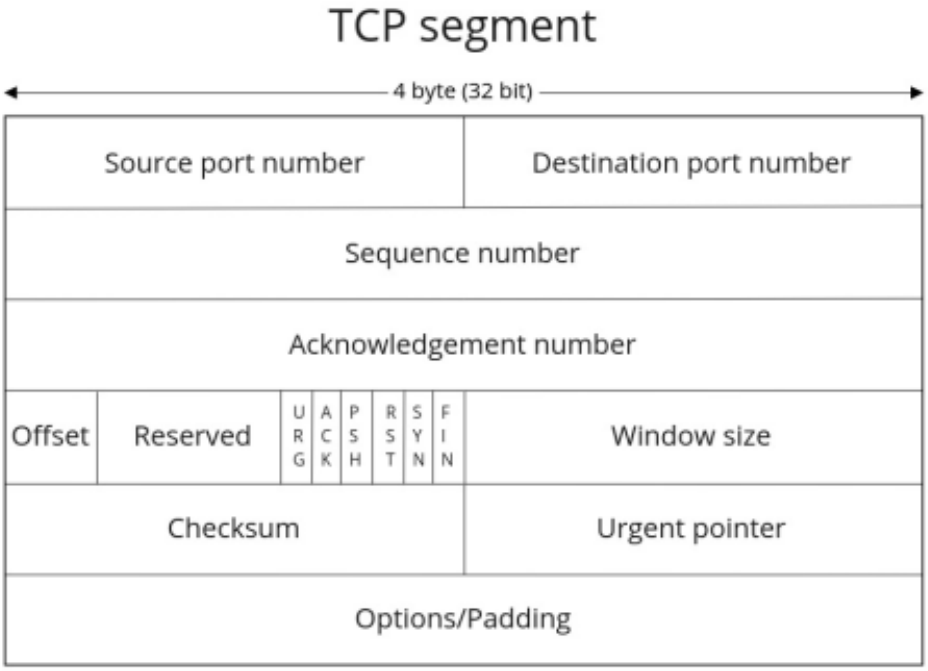


Figure 2.4: Overview of a TCP segment.

Sending packaged requires the receiver to acknowledge the package and notify the sender that the package was received. The sender sends a sequence number alongside the package. The sequence number and acknowledgement number helps both the sender and the receiver keep track of which packages were successfully transmitted and which packages to be sent again. If the receiver receives a higher sequence number than acknowledge number, a package is out of order. The receiver notifies the sender that a package is missing. Reasons for out of order can be differences in time on different routes through the internet, or a package is just lost and never received at all. Either way, the receiver has

to assemble the correct sequence from all packages received.

2.3 CAP theorem

Below is a sort description of consistency, availability and partition-tolerance in a distributed system.

Consistency Where strong consistency is guaranteed, there is a total order of operations at all sites. Although these operations are done in a distributed fashion, it appears like it is done on a single device. In a system with shared memory, it is required that all operations are handled one by one and not in parallel to avoid conflicts [4]. In a Local-first approached distributed system, consistency is weakened. Previously stated consistency is all about copies being updated and documents having the same state. For example, when the time is taken between every time connection to a network happens. If an application is based on finding peers on a local network, users must meet to get the update in a Local-first versus an application based around the cloud.

Availability In a distributed system, each node receiving a request results in a response. This means all nodes that is non-failing can send a valid response (not an error) and be received within a defined timeout. A weakness with this is how long does this algorithm run before terminating, and potential resulting in unbounded computations [4].

Partition tolerance In a partitioned system, messages sent by one part of the system are not received in the other parts of the system (communication failure). Partition tolerance means that the system tolerates messages to be lost during transport from one part to another part of the partitioned system, resulting in a system working even if it is partitioned. Communication failure can occur in a distributed system, and most systems must assume this failure will occur at some point [4][5].

CAP theorem states it is only possible to achieve only two of the three properties above in a distributed system. The theorem describes that it is impossible to develop a system that both response to every receiving request and return the expected result every time, meaning the system is either consistent and available (CA), consistent and partition tolerant (CP), or available and partition tolerant (AP).

CA means the system will output the expected results every time, and a request is always responded to. However, there is no room for the system to be parti-

tioned and hinder messages to be sent between the nodes and assure each node has the same state. Consistency and partition tolerance tolerate the system to be partitioned, and every response given is the correct one, but the time taken to get a response from the system may take time due to fixing communication failures. AP provides functionality when a communication failure occurs. Each node can respond to a request, but the node may not be up to date and respond with various results since the system can be partitioned.

2.4 Consistency

Computers in a distributed system executing data replication algorithms must eventually contain an identical copy of a shared state. As all other systems developed, the correctness of the overall system is essential, and a distributed system may handle challenges that other systems are free of, like communicating over the network. Replication algorithms come with different strength of consistency guaranteed, performance and scalability. These algorithms can be divided into strong consistency, eventual consistency, and strong eventual consistency [14].

Strong consistency A distributed system with strong consistency will optimally behave like a single node. Often, a system with strong consistency will have a node responsible for handling the total order of operations and preventing conflicts. Strong consistency may be difficult to achieve in a bigger system since a system of greater scale may want to have more than one leader and have these leaders have control over sub-parts of the system, resulting in more communication has to go through these leaders and can cause bottlenecks.

Eventual consistency A distributed system with eventual consistency guarantees, if no new modifications, that all connected devices will eventually have the same shared state. Even with a weaker consistency model, eventual consistency is able to provide better performance and scalability in both peer-to-peer and decentralised systems. This model allows changes to happen concurrently, but nodes in the system must communicate with each other since conflicting changes may happen.

Strong eventual consistency A distributed system with strong eventual consistency compromise between strong consistency and eventual consistency. Nodes that receive the same updates will achieve the identical view of the shared state even when the nodes receive the updates in different orders. Nodes in a system with strong eventual consistency solve merge conflicts by themselves and does not depend on communicating with

other nodes. This makes this consistency model suitable for a system without any central server or a leader deciding the operation order.

2.5 Conflict-Free Replicated Data Types

In a distributed system, the system needs to share data through communication of some sort and must provide a way to have the same state of elements shared on each device connected. Conflict-free Replicated Data Types (CRDT) is designed to support temporary divergence to each replica, making the data type a distributed type. Convergence is guaranteed to happen when all updates are received at each replica, resulting in the same shared state on all replicas.

CRDT solves these problems by capturing information about the updates and provides an algorithm to handle merge conflicts.

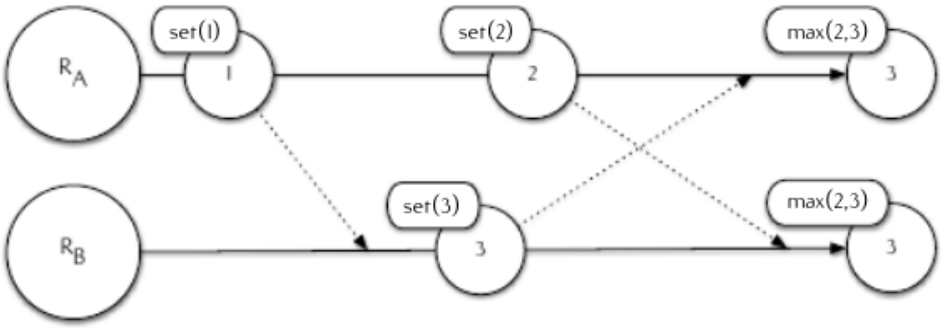


Figure 2.5: Example of how CRDT solves concurrent modification of the same element. Figure from Meiklejohn et al. [17].

Figure 2.5 shows a simple example of how CRDT solve the problem when two replicas do updates concurrently. In this example, a simple variable is incremented and a max-operation determent in which sequence the updates should be done [17].

2.5.1 State-based

State-based CRDT achieves convergence between CRDT replicas across multiple devices by distributing the entire state of an updated state from a local replica to the other replicas of the same document. The received state is then merged with the local state of the receiving device. A CRDT of type state-base is a triple (S, M, Q). The state S is a join-semilattice, which is a partially ordered set containing a join. The Q is a set of query functions returning results without

changing the state. The M is a set of mutators, which performs the updates. Figure 2.6 shows an example of how the system handle updates, and in this case user 1 is adding a key-value pair of key2: value2 to its local replica. When the replica is updated the whole state of the replica is sent/synced with the remote replicas. The receiving user merges the already existing content of the replica with the state revised.

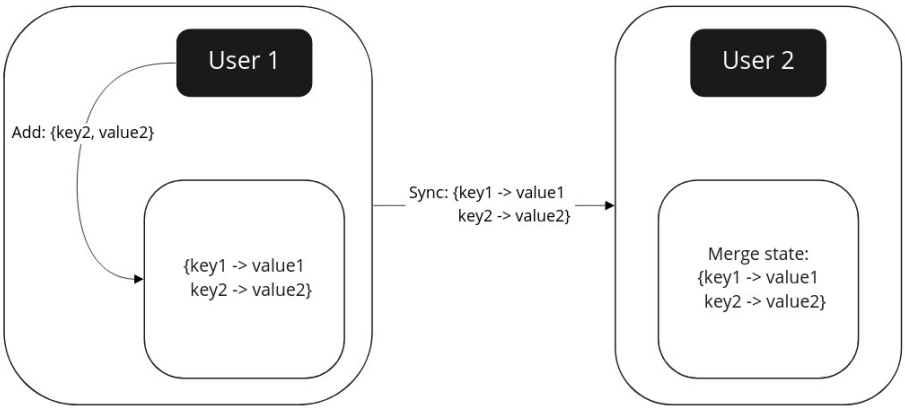


Figure 2.6: State-based CRDT example.

2.5.2 Operation-based

The same goals as in state-based CRDT are strived to achieved in operation-based CRDT. The main difference is that only the operation performed is sent to the other replicas located on other devices when a document is updated. On the receiving end, the device performs the same operation as the replica the operation originated from. In figure 2.7, user 1 adds a key-value pair of key2: value2 to its local replica. In contrast to state-based CRDT, only the the add-operation is sent to the remote replicas. In this case the receiving user does this operation to its replica.

2.5.3 Variations of CRDTs

There are many variation of CRDTs, and the list below presents some of the more common ones.

Last-Writer-Win Last-Writer-Win prioritize the last write the be done to a replica. Each write is marked with a time-stamp and the write with the latest time-stamp is the winner, independent on which order the updates are received.

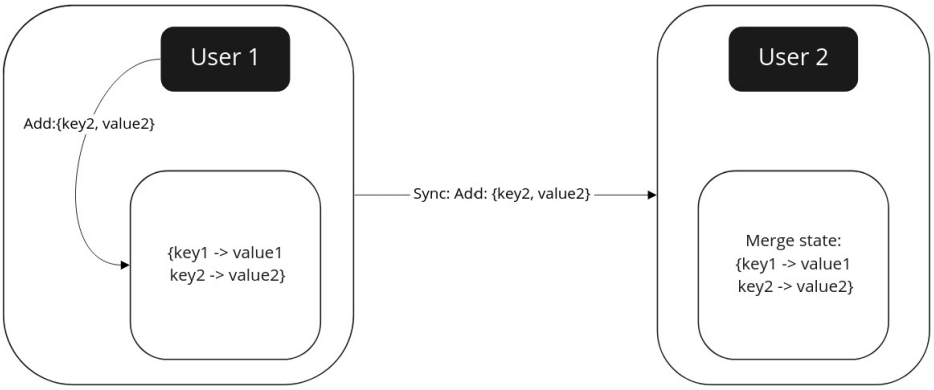


Figure 2.7: Operation-based CRDT example.

Add-wins Add-wins states that if a replica receives multiple modifications concurrent and the add-operation is then prioritized.

Remove-wins As in add-win, remove-win are prioritizing the remove-operation received.

Semantic Resolution This solution stores all updates done to each replica, and is up to the developer of the application to decide which update to be the victorious of an potential merge conflict.

2.6 Automerge

Data-centric

Data-centric refers to architectures where data is primary and are compatible with multiple applications. This means the application is built around a data model and makes all components dependent on this model (reads and writes). On the opposite, a data-driven approach is when the application handle different types of data models [18].

Model objects

Model objects in JavaScript are referred to as data-centric classes and are commonly used in almost all applications. These models can be both immutable, which means a new object has to be created when doing changes, or mutable, meaning the initial object can be changed [19].

Automerge

Automerge [15][16] is a library of operation-based JSON CRDTs used to build applications with collaboration purposes in JavaScript. The approach of building an application in JavaScript is to handle states of the application in model objects as in standard JavaScript. The library supports merging and syncing automatically and is well fitted in a Local-first application. In cases where multiple different devices, mobile- or web-application, are collaborating, each device has a local copy of the application and can do local changes to their copy. To be able to sync these local copies to have the same state is very important. Automerge is handling this situation by finding the changes done to a copy and sending a signified object to all connected devices to apply the same changes to their local copy.

Network Since Automerge is a data structure library, the chose of communication is completely up to the developer. Client-server, peer-to-peer, Bluetooth, and even USB driver sent in the mail. Automerge is also compatible with different web platforms like Chrome, Firefox, and Safari.

Immutable state An immutable object in Automerge is a snapshot of the application state at one point in time. Whenever the user make a change locally, and send or receive changes, a new state reflects the change. This makes Automerge compatible with functional reactive programming styles like React and Redux.

Automatic merging To allow concurrent changes to happen, Automerge is a Conflict-Free Replicated Data Type (CRDT), with all its functionality.

2.6.1 Documents

Document

A document reflects the state of a single Automerge instance, and are determined by a set of all changes done to it. When two documents have seen the same set of changes, even in different orders, Automerge ensures the two documents are in the same state. This implies that a document is strong eventual consistent.

Operation

An operation is a description of a single modification, and is more fine-grained than a change is. An operation changes to a single property of an object, like

inserting or deleting elements in lists.

Change

A change is defined as a collection of all operations grouped together and applied atomically. Atomically meaning all changes are applied or non of them are.

Documents are a big part of how Automerge works. After a new Automerge instance is created, the first thing to do is to create a document. It is possible to have multiple documents on the same local device and the documents only exists in memory of the device creating it and requires no networking to handle reads and writes access. To change the document, the user can either do it locally or remote; both take the old state as the first argument and returns a new state reflecting the changes.

Locally change This is the change done by the user via a user interface. Locally Automerge.change() is called, and this function groups operations that should be applied to an atomic unit.

Remote change This is when a user on another device is doing changes to their local document and the changes are sent via the network. The changes are applied to the users local document with Automerge.applyChanges(), which return a new state reflecting the changes.

Automerge has the functionality to save documents to disk. All changes done to the document are encoded to a string and stored. To load the document from the disk, Automerge decodes the string and applies all changes to a blank document. All documents stores a log containing modifications done, this log is stored when a document is saved on the device and helps to reconstruct the document when loaded into the application. Each document instance is tagged with a UUID (Universally unique identifier), and each change is numbered sequentially starting at one. This change is tagged with the UUID and numbers, making them unique and distinctive from changes done by other devices or other documents on the same device. This makes merging and applying changes with/on other documents possible.

Changes in an Automerge instance may sometimes depend on earlier changes. If an item contained in a document is created then removed by user X, but none of these changes is received by user Y, no change will be done to user Y's local document. Figure 2.8 illustrate an example where user X adds 1 to a variable a and then subtract 1 from the same variable before changes are merged with the replica user Y has, therefor user Y does not need to merge this

changes with its replica since both changes results in the same state as before the changes occur. In cases where a change will be applied to a document but are dependent on other changes not applied yet, the first change is buffered and applied in a later stage when the second change is registered.

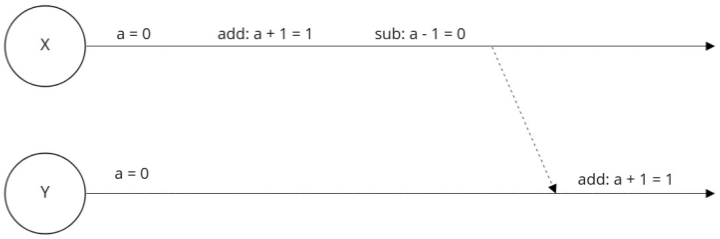


Figure 2.8: Example where changes are not merged with other replicas where these changes results in the same state as before the changes where executed.

2.6.2 Frontend-backend protocol

Automerge is split into two parts: a frontend and a backend. By being able to separate the parts, each part can be run on different threads. Frontend running on the same thread as the user application, while the backend can be run at a thread in the background.

Without the separation, the frontend communicates with the backend by calling functions directly. When separated, the backend can compute expensive tasks in the background and send the results to the frontend, and this is directly done when receiving changes from a remote device. On the other hand, user interaction is first applied to the frontend before sending it to the backend. Automerge does not handle inter-thread communication.

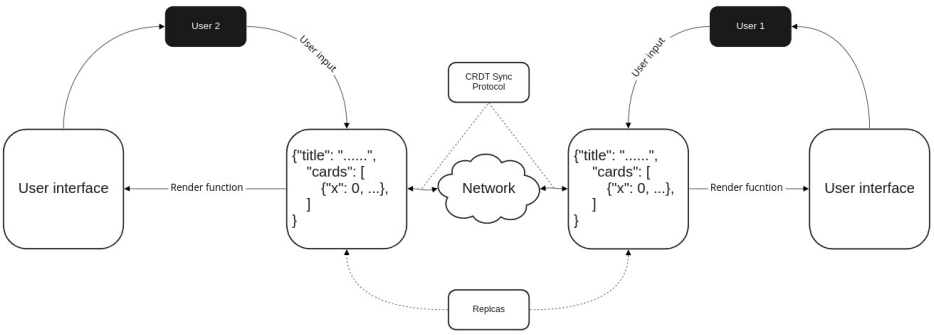


Figure 2.9: A workflow example of a system using Automerge.

Figure 2.9 shows the workflow of a system using Automerge. Each user sees

and interact with the user interface (UI), and the changes executed by the user in the UI are done on a Automerge document where the content is represented as a JSON-object. Content in t is JSON-object are used in representation of data in the user interface. When changes are completed to one document, the change is sent to other replicas of the same document over the network. CRDT, as a part of Automerge, is responsible to merge incoming changes with already existing data, and handle potential merge conflicts without communicating with the other replicas to solve this conflicts.

2.7 Tools and Frameworks

2.7.1 JavaScript

JavaScript is a high-level, lightweight programming language. This language is one of the go-to languages when developing a web application. One of many things that make JavaScript so popular is the ability to compile changes to a program or application while the program itself are running. Developing a web application often concerns small changes, and being able to run the compile without stopping the program each time a change is done makes the development process much more manageable. It is a versatile language, supporting object-oriented, imperative, and declarative programming styles, and is a prototype-based, single-threaded, multi-paradigm, and dynamic language.

JavaScript and Java is not the same language. Both are trademarks, but they share many similarities like syntax, semantics, and use [3]. All technologies concerning web or mobile development tested in this thesis are written in JavaScript, and to know the basic of usage and why JavaScript is used in libraries to be interacting with the web.

2.7.2 React

React is a JavaScript library created by Facebook. Unlike others, React is not a framework and focuses on user interfaces (UI) and how to build a UI on the frontend of the application. One aspect of React is the creation of components, which can be seen as reusable HTML elements. These components make building a user interface both quick and efficient. React combines HTML with JavaScript to create usable web applications [8].

2.7.3 React Native

React Native is a combination of native development and React. Native development means React Native provides a core set of components like View, Text and Image, and are building blocks for user interface. React renders to a native platform user interface, resulting in the application using the same platform for APIs. By having standard APIs, React can be used in development of cross-platform application, see figure 2.10, without creating platform-specified versions of components used [9].

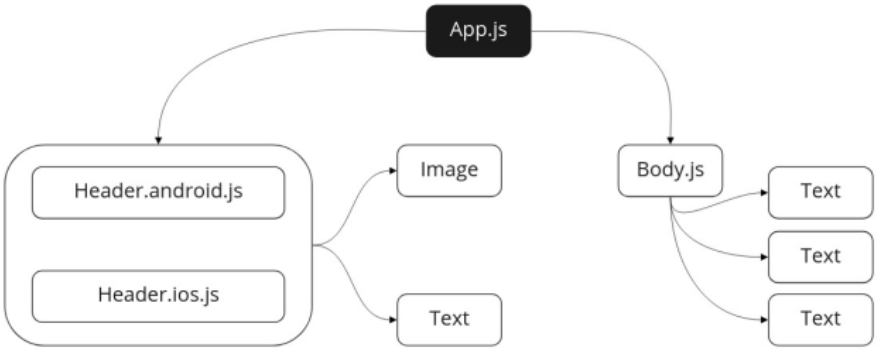


Figure 2.10: Overview architecture for corss-platform application developed in React Native.

2.7.4 Next.js

When building a web application with complete functionality, there are important details a developer has to take into consideration. How to use a webpack bundle or compilers to visualize the application, optimization of code splitting, or combining the web application with a server. A solution to these considerations are contained in Next.js.

Next.js is a React Framework. It is used in React applications for its build-in functionality like page-based routing, pre-rendering of pages both statically and dynamically, and built-in CSS and Sass support, to name a few [13].

A page in Next.js is a React Component exported from a '.js'. '.jsx', '.ts', or '.tsx' file (JavaScript or TypeScript). Each file represents a route based on its file name. In a web application, each of these files represents a new page, meaning one file is the home/start page of the application and all other files are pages the user can go to via the homepage by clicking links, buttons or other interactive components.

These pages are, by default, pre-rendered by Next.js. Next.js generates an HTML for each page in advance. By being a step ahead, pre-rendering can result in better performance, and the client-side JavaScript of the application does not need to do the rendering. A minimal amount of JavaScript code is associated with HTML, and this code is responsible for making the page fully interactive when the page is loaded in a browser [12].

Static Generation Pages using Static Generation generates an HTML at build time. This means the HTML is generated once and are used in each request for the corresponding page. For performance, this HTML can be cached. In cases where the pages do not fetch any external data, a single HTML file is generated by Next.js for each file. When a page fetches external data when pre-rendered, Next.js provides an asynchronous function that handles fetching of the external data under pre-rendering. Asynchronous means the program can request something and get a promise in responds, promising the requested subject will be fetched concurrent when other things are happening in the program.

Static Generation is default and is recommended to use when possible. Since each page can be cached, the overall performance of the application is better with Static Generation.

Server-side Rendering (SSR or Dynamic Rendering) Server-side rendering is the opposite of static rending. When SSR is used, an HTML is generated on each request received. Server-side rendering is often used in cases where the frequently updated date is needed to render.

Server-side Rendering results in slower performance than the Static Generation and are advised to only use when necessary.

Next.js has built-in CSS support. CSS, Cascading Style Sheet, is a mechanism to style web pages. This sheet contains a description of fonts, colours, spacing between objects and more [10]. Each page can have its own style sheet, or a global style sheet can be imported to each document who wants to use it. A global style sheet is often used to give a common theme for all pages connected or want to have the same layout on each page. A combination of both is not unusual since a global style sheet may only contain details about the site's frame and not colour pattern on text or buttons who occur on this page only [11].

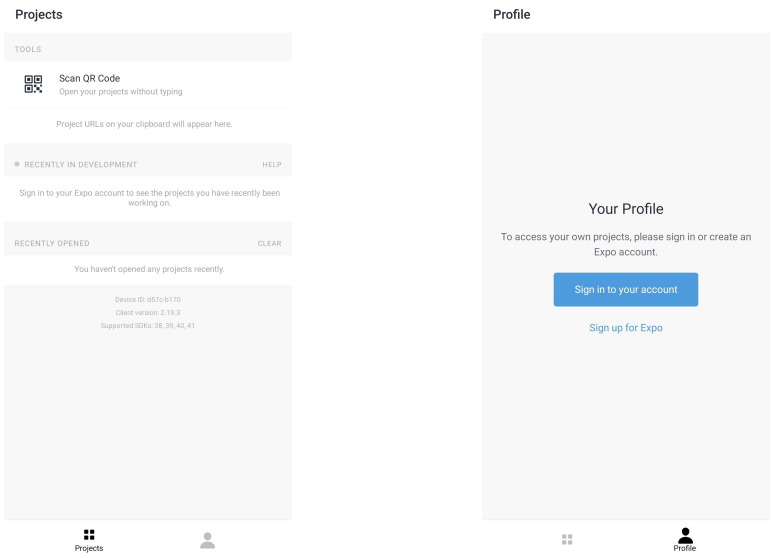
This framework in React allows developers to develop the web application only using React. Traditionally, a web application is developed with an index file in JavaScript containing functionality and an HTML file describing the applications look, but Next.js combines all this in one framework.

2.7.5 Expo

To develop an application with React or React Native, Expo is a well functional tool to use. It is a framework and a platform with tools to help the developer to develop, build, and deploy the application, independent of which platform the application is meant for. Android, iOS, and web uses the same codebase of JavaScript, or TypeScript [23]. Expo is a versatile platform and fits both new and experienced developers with options to start with a bare canvas of an application or an application with more content for the user use as an example of how to use React or React Native. This platform provides functionality like a "plain" application with just JavaScript or TypeScript, build Android or iOS builds, over the air update function, to name a few [24].

Although the many advantages to using Expo, there are some limitations. Some of the limitations, like not all iOS and Android APIs, are available and a minimum cap of OS versions. Expo has some requirements towards the standard of technology of the devices used in development. Services like free build and the size of the build are limited and may not fit other requirements for the application like scalability. The last limitation is set towards developing applications only targeting children under 13 years old due to strict guidelines for Apple and Google, who owns platforms to download apps [25].

Figure 2.11a and 2.11b show the Expo app. Users of Expo can create a profile to store projects the user has created or are a part of. The home screen provides multiple choices; scan a QR-code, select a project stored in profile or open a recently opened project. When a Expo program is run in a terminal the QR-code appears in the terminal, or when testing already existing programs the QR-code can be accessed if the webpage or repository provides it.



(a) **(b)**

Figure 2.11: Home screen of Expo mobile app (a), with possibility to make a user (b).

3 Design

This chapter gives an insight into the design choices for this application. Section 3.1 will describe the overall view of the Local-first application, section 3.2, 3.3, and 3.4 will go into more detailed choices of each main part; server, web browser, and mobile.

3.1 Overall view

The overall architecture of the prototype developed in this thesis is displayed in figure 3.1. It is developed two different but relatively similar codebases; one for mobile devices and one for computers. The codebase for the mobile does not differentiate between frontend and backend, meaning all functionality is run on the same thread, and no internal communication is required. Computer code is separated, resulting in two internal code bases; one for the server and one for the web browser. For a user to interact and use the prototype in a web browser, the user has to run the server —more on how and why later in this chapter. This architecture allows multiple mobile devices to connect to a server. The computer hosting the server can have multiple frontend instances in different web browsers and tabs within the same web browser as long as it is on the same computer as the hosted server.

Devices connected through the server are able to modify Automerge documents concurrently. Automerge provides the functionality when modifications are

done, like adding or deleting a document and adding, deleting or modify content in an Automerge-document. Each device in this prototype has a set containing all Automerge-documents the device is a part of. Since Automerge provides strong eventual consistency, it is possible to have replicas exchanging messages with modifications and each device handling merge conflicts, if any, on its own.

Each mobile device and web browser instances have the option to create a document. A newly created document has a title, and the rest are empty. This empty space in a document can be filled with cards containing a question and a set of answers. Since the application is Local-first, these changes can be completed without a network connection, and the users can geographical be anywhere as long as the device has power.

A potential scenario where this prototype can be used is in combination with lecturing. The teacher or lecturer runs the server/web browser instance, and students or participants are connected to the server with their mobile devices. The teacher/lecturer creates an empty document fore the students/participants to add questions to during the lecturing. The prototype allows both the teacher/lecturer and follow students/participant to answer the questions asked, so if a fellow participant knows the correct answer, there may not be necessary for the lecturer to answer. However, if an answer is incomplete or does not have the right answer, there are possibilities to add more than one answer. Automerge, with strong eventual consistency, will in this scenario make all node see the same order of questions and the same order of answers, making it identical on each device.

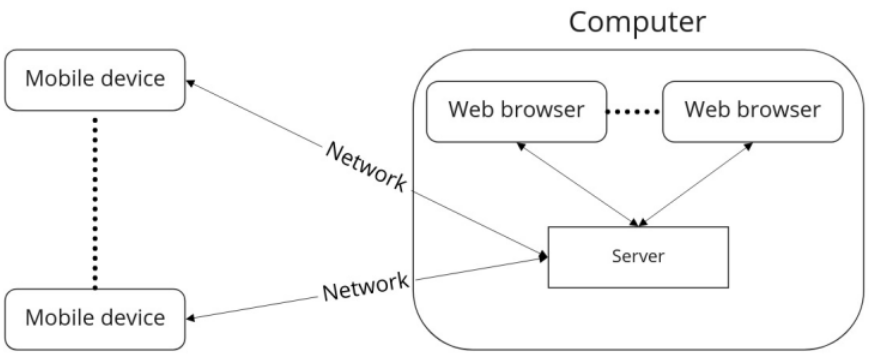


Figure 3.1: The overall view the system.

3.2 Server

The server is a dynamic server in a server-client approach. The idea behind a dynamic server was to enable the possibility to use the system without a connection and be dependent on other systems to work and be available the times the user wants to access its documents. In this prototype, the server serves as a middleman for communication, meaning all clients, both mobile devices and the frontend in a web browser, has to go through the server to communicate with other clients. A client sends a single message with a description or a keyword to the server, and the server will then broadcast this message to all devices connected except the sender of the original message.

To interact and use the forntend in a web browser, the server must be initiated. The server fetches and routes all Next.js pages to be used in the browser, and one of these pages serves as the home screen or home page of the prototype. There is no requirement to have an internet connection to initiate the server, but none of the communication functionality is available. If a connection possibility is detected, the server will automatically connect, and all functionality is restored.

The server is developed with a combination of Socket.io, Express and Next.js. Socket.io is a JavaScript library and is used for networked application and make it possible to communicate between the server and the clients. To enable this communication, Socket.io is divided into a server-side library and a client-side library that creates bidirectional channels between each with a WebSocket connection. Express is a framework to help develop backend web applications, and does the routing and have the responsibility to respond to incoming requests. Furthermore, Next.js is a framework to develop frontend applications but does have functionality, as explained in 2.7 under Next.js, for server-side rendering.

Next.js is the server application, Socket.io is the communication channel, and Express is how and what to respond to requests from clients.

3.2.1 Alternatives

Server-client is an approach to handle communication between devices, and to change this method will also change how communication works. Another alternative is more of a direct approach to communicate, meaning a server is not necessary. A possible alternative is peer-to-peer. With the use of the internet or other wireless technologies to handle communication, this alternative is the closest to achieving the goal of collaboration. Peer-to-peer is a direct-connection oriented technology, meaning two devices can connect directly to each other

as illustrated in figure 3.2, and only depend on both devices having access to the same technology.

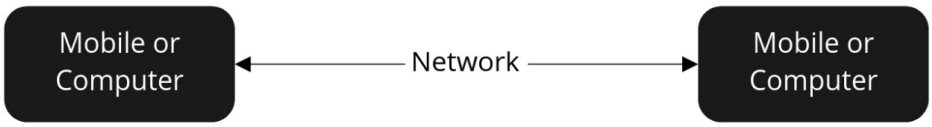


Figure 3.2: Peer-to-peer system.

In a small system where networking is minimal, a server seems unnecessary to initiate each time a connection has to be established. In this case, a direct oriented connection is the better option since a central component is not needed. The opposite happens if the system depends on communication with tens of devices. A device must then route the message itself and find the closest device it knows to the destination to forwarding the message to. Each device to receive a message must do this process for the system to work.

3.3 Web application

The web application is the frontend of the prototype run on a computer and is rendered in a web browser for the user to interact with. There is only one web page at the moment that serve as the homepage, so there is minimal to route for the server, but it is possible to add new pages to the prototype without changing the backend. This end is developed with React and has the client part of the Socket.io library, Automerge, and Next.js. Socket.io creates a WebSocket to be able to connect and communicate with the server-side Socket.io. Automerge is used to handle creating, deleting, merging and changing documents as a reaction to what the user chooses to interact with on the web page and handles possible conflicts on its own. Next.js provide additional HTML functionalities to standard HTML.

The user has a few input fields and buttons to interact with on this web page and is displayed in figure 3.3. As stated in section 2.6, to create, delete, or change a document, the corresponding documents name must be known to handle these operations. Therefore, each row of input/s requires a name. All documents created locally or remote are displayed in "Documents", a JSON representation of the documents.

Operations like creating or deleting a document do not require other information or input than what to call the new document or the name of the document to be deleted. The next operation is adding a question to a document. Here

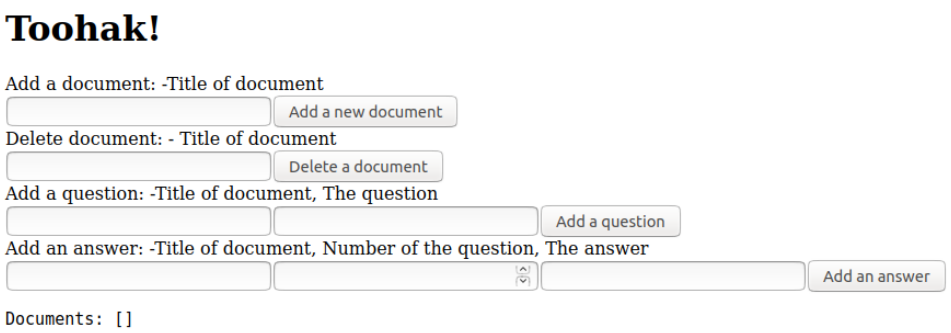


Figure 3.3: Home page in web browser

the first input is the name of which document to add to, and the second input is the question to be inserted into the Automerge document.

The last operation requires more of the user to complete. Adding an answer to a document requires the document name, an index number, and the answer. Note that a question can have more than one answer.

Figure 3.4 represents a document with title "Test 1". In this prototype, the name of the document is also represented as the documents ID. This document has a list name "cards", which contain all questions and their answers. The question "What is CRDT?" has index 0, so the user has to type 0 into the "Number of question"-field in figure 3.3, to add an answer to this question. This question has one answer, "Conflict-free Replicated Data Types". If "Test 1" receive a new question, this question will have index 1.

3.3.1 Alternatives

The homepage can be seen as crowded with information when many documents are created, with multiple questions and answers added. An alternative may be filtering out the title of documents and making them clickable, directing them to a new page containing content from the selected document. To list these titles in sequential order will make it easier for the user to get an overview of which documents represented. By pages only containing the content of single document information will be easier to retrieve and modify.

In a web browser, the user interaction design is the easiest to change and has to most options. Technology chose to render the page and handle operations is mainly chosen between using React or an index/HTML combination. React is more of an object-oriented approach with classes and components which can be reused, and a web page can be developed in a single document. Index/HTML

```
Documents: [
  [
    "Test 1",
    {
      "title": "Test 1",
      "cards": [
        {
          "id": "133febca-701f-4115-8846-c93a878b4123",
          "Question": "What is CRDT?",
          "Answers": [
            {
              "id": "3b75635a-97f9-46e4-a661-8014ce0facf2",
              "a": "Conflict-free Replicated Data Type"
            }
          ]
        }
      ]
    }
  ]
]
```

Figure 3.4: Representation of a document set in JSON.

combination has an index file written in JavaScript handling functionality and computations, and HTML file handling the visual aspect of a webpage.

3.4 Mobile application

The prototype running on a mobile device is developed in React Native, with its own Socket.io and Automerge. Socket.io provides functional communication to the server-side Socket.io. Automerge is used to create, delete, merge, and change documents to react to what the user chooses to interact with in the mobile application. The mobile application has a simple look, the homescreen is shown figure 3.5a. With only two buttons for the user to use, there is minimal interaction on this screen. The button on the top right is to connect to a server, and when used, the user gets a pop-up window to insert the IP address to the computer the server is run on. The inserted IP address is displayed under the prototype's title to remind the user on which IP the user is connected to. There is no check for if the inserted IP is valid or the inserted number is formatted as an IP address and the prototype will try to find a Socket.io instance listening on the inserted IP.

The other button is located at the bottom of the screen to make it easier to reach on a touch-screen since adding a new document is used more frequently

than connecting to a server. Adding a new document is also achieved inside a pop-up window, see figure 3.5b. Only the name is acquired to create the document. Using a pop-up window, the window functions as an alert for the user that the user itself is about to do an action. The list appears in the middle of the screen; see figure 3.5a for reference, and is scrollable inside a designated area of the screen, making the list easy to access. The button to add a document does not follow the list and disappear down the screen. So even if the user has multiple documents created or newly load the screen, the user does not need to scroll down to the bottom of the list to find the button. Figure 3.7a illustrate this feature.

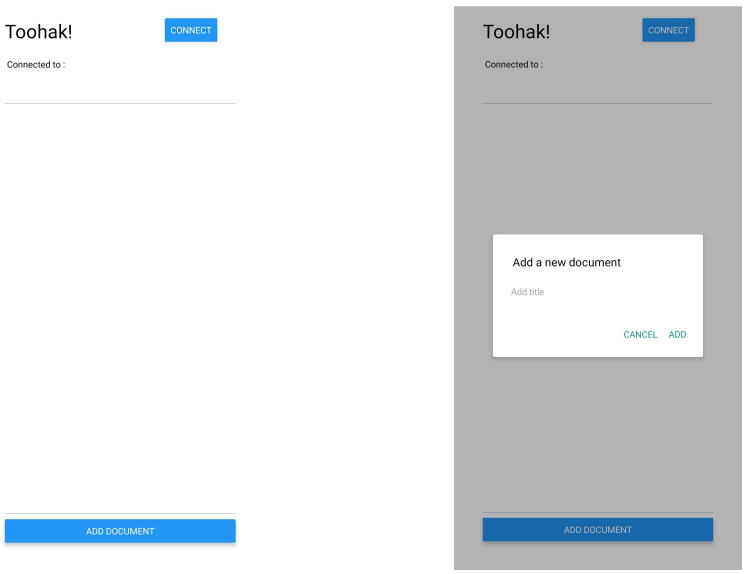
What the user sees in the second screen depends on which document -item in the list- chosen, figure 3.6a show a new, unmodified document. The only thing the user can do is add a question -a card- to the selected document. The button and list on this screen are identical to the homescreen. Figure 3.6b shows the pop-up window the user gets when adding a new question. The list has its area to render in, and the button is at the bottom of the screen. Each question in the document is displayed as cards with the question as the title of the card at the top, a list of answers, if any, in the middle, and a button to add an answer to the question on the card, figure 3.7b shows a card with a question and two answers. Answers are appended to the list of answers by inserting the answer in the input field displayed on the card and by pressing the button on the card.

Since the mobile application has multiple screens the user can interact with and require access to the network, there are components located in a common shared file. This file contains the communication functionalities and holds multiple shared variables to be accessible.

3.4.1 Alternatives

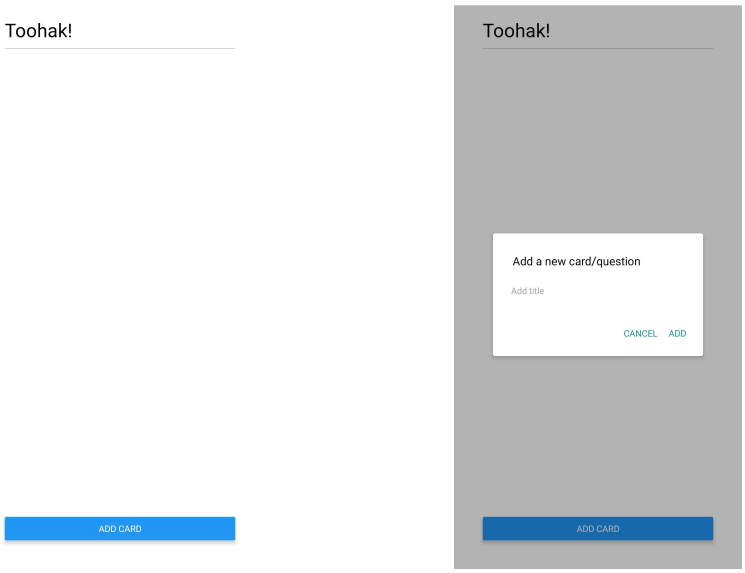
Alternatives for the home screen are to make it even more straightforward than it already is by making the document list and functionality to add a new document to its own screen. This makes it more distinct to functionality added in the future and makes it an option to let users categorise or put documents in groups shared with different people.

Technology options for developing a mobile application are many. It is more of a preference for the developer to choose the technology that suits best and provides the necessary requirements for developing and deploying a mobile application.



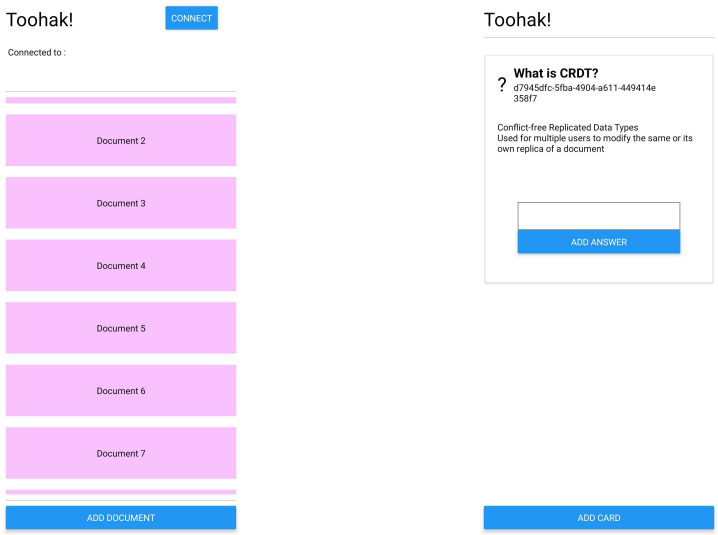
(a) **(b)**

Figure 3.5: The home screen (a), and the pop-up window to add a new document (b).



(a) **(b)**

Figure 3.6: Screen to display content of a document (a), and the pop-up window to add a new question (b).



(a) **(b)**

Figure 3.7: Home screen with documents added to the list (fix this) (a), and a question displayed in a document (b).

4

Implementation

This chapter describes the implementation details of the prototype. The prototype is used to demonstrate the functionality of Automerge and how Automerge fits in a distributed application with a Local-first approach and requirement of collaboration and offline as a non-failure state. All parts of this project are developed in JavaScript. Section 4.1 gives a description of the server, 4.2 the part shown in the web browser, 4.3 the application running on the mobile device, 4.4 describe common components independent of which device the prototype is running on.

4.1 Server

The server is the backend part of the web application and consists of two parts. Part one is a preparation phase, and part two is the functional server running when the web application is running. The preparation is a sequential section of importing the necessary libraries; Express, Next.js, and Socket.io. The next step is to set a boolean to decide if the server is to run in developer mode or not. Since this is a prototype, the system is run in developer mode for enabling hot-reload. Then a request handler from Next.js is initiated; this handler is a default function handling incoming messages and requests for the server. Part two of the server is the app. The app returns a promise, which allows the developer to use a then-function to attach a pipe—the then-function initiate Express, which fetches all routs and returns them to the request handler. Ex-

press uses pages, a page is a site in the web application, and when routed, the application can fetch the correct page when the page is up to show, listing 4.1 is the code to achieve routing.

```
const server = express()

server.get('*', (req, res) => {
    return handle(req, res)
})
```

Listing 4.1: Server to fetch and route all web pages in the system.

Communication is provided by using socket.io; see listing 4.2. Socket.io establishes channels between the Socket.io server and the clients with a WebSocket connection and uses HTTP as a backup. WebSocket uses a TCP to establish a handshake with the client. The server only listens on a specified port, and the client is responsible for initiating the handshake process. When connected, the client can get the server to forward messages with a specified keyword or a short sentence to describe the purpose of the message or what the server must do.

In the pipe, multiple cases are covered. Listing 4.2 shows a io.on function which applies all corresponding functions to a socket when connections are established. Socket.on functions are input-dependent, meaning which functions called are dependent on what the user chooses to do. There are multiple socket.on instances. Each one of them is triggered by the "\\message or instruction" input parameter. "\\parameters" is a set of inputs; the title of a document and changes in string-form if the message concern modifications.

```
const sio = require('socket.io')
const http = require('http').Server(server)
const io = sio(http)

io.on('connection', (socket) => {
    socket.on('\\message or instruction', ({\\parameters}) => {
        \\operations
    })
}
```

Listing 4.2: Server start with imports of libraries and start a socket to handle communication.

4.2 Web browser

The web browser application is developed in React, and the homepage is a React class component. By being a class, if necessary, the same page can be replicated and reused.

This class has a constructor that contains states of pre-set variables and keeps track of states. This constructor also binds functions within the class to be used inside the class itself. React classes has an included function that can be used to set variables or start WebSockets. This is a componentDidMount() function which is called when the class is first run. By initiate and mount a WebSocket instance of Socket.io, it will provide communication between the server and the web browser without bothering the user to deal with it. Furthermore, the class contains functions to add and delete documents, add cards -question cards- to a document, and the user can add an answer to a question. For more detail on these operations, see section 4.4.

Like all other web pages, HTML is required for the page to be rendered for the user to see. On the page, the user has four rows of inputs and buttons. The first row is inserted name and button to add a document, and the second is a name and a button to delete the document. The third is the name of the document, title of the question and button to add the question to the document, and the last row is the document name, index number of the question, the answer to be inserted, and the button to add the answer.

A privilege the user of a web browser instance of the application have is to delete a document from the document set, shown in listing 4.3. Automerge provides the function for deleting the document, but the document ID has to be provided. The ID in this system is the name of the document.

```
let doc = docSet.getDoc(this.state.docname)
if (doc) {
    docSet.removeDoc(this.state.docname)
}
this.setState({docarray : Array.from(docSet.docs)})
```

Listing 4.3: Deleting a document.

4.3 Mobile application

The mobile application is developed in React Native, and consist of one homescreen and on dynamic screen. Both screens are represented as classes, the same as the web application is a class, and can be reused if necessary.

The homescreen has a constructor containing states used inside the class. Since React and React Native is similar a componentDidMount() function exists, but since the application is an independent application and can be used on its own, there is not always an available server to connect to. Instead, the class contains a function to manually connect to a server through a pop-up window at the homescreen. This function is run when the user insert an IP address as input and tries to create a connection between the mobile and the server. If successful, a client instance of Socket.io is established in a shared file and the mobile user can communicate with the server through WebSockets.

This screen has two render functions, one for rendering components shown regardless of whether the application are newly installed or been used for a while, and one for rendering the document-list in the middle of the screen which is dependent on user-input. Each document created is displayed as independent items in the list and are clickable through the native components React Native provides. The homescreen class only contain functionality to create a new document and insert it into a shared set containing all Automerge documents, all other operations is dedicated to happen inside a document.

The second screen is dependent on which item the user select form the homescreen. This class also contains a constructor with states in, but in contrast to the homepage, do have a mounted function handling parameter sent to this class, which in this prototype is only the name of the document selected. Document name allow the application to fetch the document from the shared document-set the screen corresponds to, and insert questions and find the right question to add an answer to, see section 4.4 for more information about these operations. Two render functions are also used on this screen. The first renders all static component, and the second function render the cards created when a user inserts a question. These cards contains a question and a list of answers. To add an answer the user must use the text-input corresponding to the question-card.

Both screens interact with the network and the same set of variables, so variables and the WebSocket is in a shared file. This makes them accessible for both screens without creating a loop of imports between the two classes representing the two screens. This shared file contain the document-set where all created documents are stored, and the Socket.io instance to handle incoming and sent messages for the whole mobile application.

4.4 Common components

This subsection describes three common components. The components are the same for both web and mobile, and are functionality for adding a new document, adding a new question and to add an answer to a question.

4.4.1 Add a new document

Add a new document to a document set is shown in listing 4.4. Since a document is immutable, changes can not be done directly to the document, so the change must be done through a change function from Automerge. The change function is called with a callback, meaning the state of the document can be mutated. As shown in the 4.4, a new document is changed, to begin with, but the change function takes a document as an input and returns a document. In this prototype the change-function provided by Automerge creates a new document instead of taking an existing one. When a document is created, a change happens right away by changing the empty document to contain a title, representing the document's ID and an empty array to store cards.

A DocSet is a set containing all created documents, and for a document to be contained in the set, the document has to be set. This set function takes an ID and a document as input. The document is mapped with the ID, so when the user is making changes to a document, the ID (documents name) must be known before changing it. For the changes to mitigate to the other connected replicas, the changes as to be detected and sent. Automerge has two ways of detecting changes. The first one is getChanges function. This function takes the old document and the document generated from Automerge.change, and will only detect the differences between the two documents. The other way is getAllChanges, which detect all changes ever done to the document, from the creation of the document to the latest change done to it.

The changes are then stringified, from a JSON object to a simple string, to be sent along with the document's name. Socket.emit sends the description, "add doc", and a set with the title and the changes. In a web browser, this emit function is always called because the server is initiated alongside the web application. On a mobile device, the user has to connect to the server and thereby must have a socket created for the device to communicate with the server. Therefore a simple check is done to see if the socket exists or not.

```
const initDoc = Automerge.change(Automerge.init(), doc =>
    {doc.title = this.state.docname, doc.cards = []})
docSet.setDoc(this.state.docname, initDoc)
```

```
this.setState({docarray : Array.from(docSet.docs)})

const changes = Automerge.getAllChanges(initDoc)

this.socket.emit("add doc", ({title: this.state.docname, change:
    JSON.stringify(changes)}))
```

Listing 4.4: Adding a new document.

4.4.2 Add a question

The next common functionality is adding a question to a document, see listing 4.5. Each document has an array to store cards. These cards are JSON objects with an ID, the question, and an array to hold answers. Since questions can be of the general type, multiple answers can be submitted to the same question. The ID is a UUID and are used as IDs for listing and allowing the same question to occur multiple times without conflicting with each other. To get the right document to change, the document's name must be provided to the function. A check is done to see if the document the user wants to modify exists, and if it exists changes and networking are completed. Since each document has a card array, the new JSON object is added to this array.

In contrast to adding a new document a change only takes an already existing document as an input and returning a new document with the updates. The update and getAllChanges is done in the same fashion as adding a new document.

```
let id = uuid.v4()
let doc1 = docSet.getDoc(this.state.docname)
if (doc1) {
  const doc2 = Automerge.change(doc1, doc => {
    doc.cards.push({id: id, Question: this.state.question, Answers:
        []})
  })
  docSet.setDoc(this.state.docname, doc2)
  this.setState({docarray : Array.from(docSet.docs)})
  const changes = Automerge.getAllChanges(doc2)

  this.socket.emit("add question", ({title: this.state.docname,
      change: JSON.stringify(changes)}))
}
```

Listing 4.5: Add a question to a document.

4.4.3 Add a new answer

The last is adding a new answer to a question and are shown in listing 4.6. Each card has a designated array for answers under the question. Finding the correct document and to give a UUID to the answer is identical to the way adding a question is added. The ID is used for allowing the same answer to be added without conflicting with each other and are used as an ID for list elements.

```
let id = uuid.v4()
let doc1 = docSet.getDoc(this.state.docname)
if (doc1) {
  let doc2 = Automerge.change(doc1, doc => {
    doc.cards[this.state.questionNumber]['Answers'].push({id: id,
        a: this.state.answer})
  })
  docSet.setDoc(this.state.docname, doc2)
  this.setState({docarray : Array.from(docSet.docs)})
  const changes = Automerge.getAllChanges(doc2)

  this.socket.emit("add question", ({title: this.state.docname,
      change: JSON.stringify(changes)}))
}
```

Listing 4.6: Add a new answer to a question.

5 Evaluation

This chapter will discuss testing, both locally and over the network. Design and implementation chooses is discussed in Discussion 5.2, with personal experience in 5.2.1 and ideas of future work in section 5.3.

5.1 Testing

Testing is an essential factor in understanding how the prototype works practically and the tests targets underlying technology to achieve a set of goals. Multiple technologies like Perge, Hypermerge and Hypercore, as well as current state of the prototype with WebSocket from Socket.io, server functionality from Express, and HTTP for listening clients in a server-client approach.

5.1.1 Test devices

All tests, both locally and over the network, was tested on the same two devices. Perge, and Hypermerge and Hypercore was tested on the computer. The current prototypes technologies; Next.js, Express, and Socket.io was tested on both devices. The devices used in testing was one computer and one mobile; they have the following specifications.

Desktop:

- HP EliteDesk
- Intel duo quad-core i7-4770 CPU
- 15.6 GiB System memory
- Ubuntu 18.04.5 LTS

Mobile:

- Samsung Galaxy S8
- SM-G950F
- Android-version 9

5.1.2 Test methods

Two types of testing were directed to test the functionality and flexibility of the system. All testing is done manually, meaning all tests were done in person, testing operations were done one by one, and the connection is connected by manually inserting the IP address.

Offline testing

Testing done on all technologies was first tested locally to make sure the basic functionality of adding a new document, adding content to a document, modifying the already existing document, and the possibility to extend the already existing project.

Online testing

Over the network, all functionalities tested offline was tested with concurrent modification. This part of testing focuses more on handling communication and finding pros and cons with the network technology used. Online testing did also test how Automerge handled concurrent changes, and are tested with scenario described in 3.1 in mind.

5.2 Discussion

Communication methods can vary since Automerge is agnostic to the network, and multiple approaches are viable. Three different technologies were tested to decide which technology fit the current idea and goal for this thesis. The three technologies tested was Perge, with connection over PeerJS, Hypermerge and Hypercore with peer-to-peer connection, and the current implementation with a dynamic server-client approach.

Perge is a library for developing a distributed system where each instance is located in a web browser and uses peer-to-peer communication. This approach is useful for developing a system only located in the web browser, and Perge provides a way to make a user and to store information about connected users. An example developed by the creators of Perge illustrates peer-to-peer technology and Automerge working together to achieve collaboration; however, this example is minimal and supports only local nodes communicating with each other. Changes done by local nodes is handled by Automerge and are outputting the expected results. The idea of having a local user that can store information about other nodes was the preferred way of handling communication and user info in the early stages of developing the prototype. However, Perge does not provide cross-platform communication.

Hypermerge in combination with Hypercore and Automerge is demonstrated in PushPin and Pixelpusher. Hypermerge is a Node.js library to develop collaborative applications with peer-to-peer communications, and Hypercore is distributed append-only log for logging modifications done to documents. PushPin is developed in React, demonstrating that Automerge and React work for online and offline web usage. Another already existing project with Automerge, Hypermerge and React is Pixelpusher, and uses modern web technology but is software installed locally. Pixelpusher is software to create pixel art, and multiple users can work on the same art through a peer-to-peer connection. This technology was to prefer at the start of this project, but a mobile device needs to have all necessary libraries installed to the full extent when installed. Some libraries are dependent on other libraries, and some of these libraries do not exist anymore, making the installation fail due to trying to install non-existent libraries. So these two examples do prove that Automerge-functions are suitable for cooping with the network but not suitable for mobile devices. This combination can be used on web applications or programs developed in React for usage on the computer, but this combination is not viable in React Native and mobile applications.

Due to experiments with different already existing combinations of technologies with Automerge, the current solution was to use a new combination of technologies to develop a Local-first application with Automerge used for col-

laboration over the network. Part of the solution was to develop server-client communication. This opened the doors to use Socket.io and Next.js, which could provide functionality to both the server and the clients, and resulting in similar, but separated codebases, to interact with each other. Some of the components are identical in both codebases as shown in section 4.4. A server-client approach uses HTTP and TCP to handle communication. HTTP used for fetching the necessary resources to complete the webpage rendered in a web browser and to have the possibility to develop the web pages further to have more content. Since this kind of application depends on messages to be delivered completely, having a transport protocol that allows the message packages to be lost or delivered out of order and to provide a solution to these possible problems are important. TCP, by establishing a connection to the receiving node, have the possibility to send lost packages again and have control over the sequence the packages should be assembled in. UDP does not provide what TCP does and therefore not used in this prototype to handle communication.

The current state of the prototype was tested sequentially. Starting with creating a document, adding a question, and adding an answer to a question locally first and evolving to testing with multiple devices over the network. With the prototype distributed on different types of devices, doing tests locally is essential to get each part to work correctly before testing it over the network.

Making the computer do the server-work and creating a server instance when needed instead of depending on an external server to handle communication was key to both performances and to achieve the essentials of a Local-first application. The servers only function is the receive and distributes messages. Since the server is the backend and the web interface is the frontend of the computer application, the server has to send messages to the web, although it is on the same device. Whenever the server gets a message, this message is forwarded to all connected devices in the form of a broadcast.

The tests were completed with the scenario in 3.1 in mind. Automerge preform as expected when it comes to displaying added content in identical order on both test devices. When two changes are done concurrent, Automerge will arbitrarily take one of the changes and place it before the other change, and applies to both adding questions and adding answers to the same question simultaneously. This is important to achieve because of how users of the prototype in web browser depend on which index the question is located on. By having the questions in the same order, answers are added to the right question on all devices.

5.2.1 Experience

My personal experience with researching and finding different technologies to complete a functional prototype has been educational and opened an understanding of more in-depth research, not only to get a result but also to understand independent work, planning, and methods of working. This has been four and a half month to plan, and to have to manage this time to complete a project of a greater scale has been educational.

There have been different learning curves depending on the technologies found and researched. To learn JavaScript without experience of using it before was a challenge, so for using already existing communication libraries without any previous experience with programming related to the network. On the other hand, React, React Native, and Automerge have been easier to learn and understand because these libraries are very good documented and to get a general understanding of how these libraries work do not require additional knowledge of other libraries.

5.3 Future Work

Potential future work for the prototype is user account opportunity and to be able to create different rooms or groups. In the current state of the prototype, each part of the system does not have any possibility to make an account. The prototype, by default, does only have security on the connection channel between devices. An account or user interface will make it possible to verify the users and have control over which users they are connected to.

Furthermore, with an account, making groups is a possibility. The current system is designed for users to only work on the same theme of work, for example, a document per lecture in the same course. To categorise courses in different groups with different people represented in each group and for people to be represented in multiple groups without syncing documents the user does not want or are part of.

Scalability and usability are yet to be explored and are essential factors to consider when evolving the system running on the prototype. Since the prototype is tested on two devices, there are no concrete results that imply that the system works on tens of devices simultaneously. Usability is also yet to be explored; what can this prototype be used for? Who does this type of system fit? Which profession do this kind of application potentially be used by?

There is also space for exploring different technologies and finding one or

multiple that suits the system better. Does this prototype work in a more significant setting? Furthermore, will the current technologies be maintained in the future?

/6

Conclusion

In this thesis, I have explored different technologies to develop a Local-first application that can run on different types of devices for collaboration. Different technologies are required to get devices to communicate and collaborate over the internet, resulting in testing libraries and other prototypes like Perge with PeerJs, and Hypermerge, a peer-to-peer library in combination with Hypercore, a library for a distributed append-only log. Peer-to-peer and client-server communication has been explored and tested, where UDP, TCP, or HTTP functionality where taken into consideration. Various techniques and implements have been tested to understand better what technologies provide the necessary features to complete the goals. The result is a Local-first prototype developed in React and React Native with Next.js, Socket.io, Express, and Automerge for collaboration purposes and have a server-client approach to communication.

Bibliography

[1] Kleppmann, M., Wiggins, A., van Hardenberg, P., and McGranaghan, M. Local-first Software: You Own Your Data, in spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN In-ternational Symposium on New Ideas, New Paradigms, and Reflec-tions on Programming and Software (Onward! '19), 2019.*

[2] Basilis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J., and Stoica, I. Highly Available Transactions: Virtues and Limitations (Extended Version). In *40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.Proceedings of the VLDB Endowment,Vol. 7, No. 3, 2014*

[3] MDN Web Docs What is JavaScript? *https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript. 2021*

[4] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *SIGACT News, vol. 33, no. 2, pp. 51–59, 2002. 2002*

[5] Kleppmann, M. A Critique of the CAP Theorem In *arXiv preprint arXiv:1509.05393. 2015*

[6] Brewer, E. A. CAP twelve years later: how the "rules" have changed. In *IEEE Computer, 45(2), pp. 23–29, 2012*

[7] Shapiro, M., Nuno M. Preguicca, Baquero, C., and Zawirski, M. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, pp. 386–400. SSS 2011*

[8] Rascia, T. React Tutorial: An Overview and Walkthrough. *https://www.taniarascia.com/getting-started-with-react/. 20.08.2018*

[9] Facebook Open Source React Native. *https://reactnative.dev/. 2021*

[10] W3C What is CSS? *https://www.w3.org/Style/CSS/. 2021*

[11] Vercel Built-In CSS Support *https://nextjs.org/docs/basic-features/built-in-css-support. 2021*

[12] Vercel Pages *https://nextjs.org/docs/basic-features/built-in-css-support. 2021*

[13] Vercel Create a Next.js App *https://nextjs.org/learn/basics/create-nextjs-app. 2021*

[14] Gomes, V. B. F., Kleppmann, M., Mulligan, D. P., and Beresford, A. R. Verifying Strong Eventual Consistency in Distributed Systems. In *Proc. ACM Program. Lang., Vol 1, No. OOPSLA, Article 109. 2017*

[15] Kleppmann, M., et al. Automerge. *https://github.com/automerge/automerge. 2020*

[16] Kleppmann, M., et al. Automerge internals. *https://github.com/automerge/automerge/blob/main/INTERNALS.md. 2020*

[17] Meiklejohn, C., and Van Roy, P. Lasp: A Language for Distributed, Coordination-Free Programming. In *PPDP '15: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming. 2015*

[18] McComb, D. The Data-Centric Revolution: Data-Cantric vs. Data-Driven. *https://tdan.com/the-data-centric-revolution-data-centric-vs-data-driven/20288. 2016*

[19] Java Practices 3.011 Model Objects *http://www.javapractices.com/topic/TopicAction.do?Id=187. 2018*

[20] MDN An overview of HTTP *https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. 2021*

[21] Fox, P. User Datagram Protocol (UDP) *https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/user-datagram-protocol-udp. 2021*

[22] Fox, P. Transmission Control Protocol (TCP) *https://www.khanacademy.org/computing/computers-and-*

internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/transmission-control-protocol–tcp. 2021

[23] Expo Introduction to Expo. *https://docs.expo.io/. 2021*

[24] Expo Workflows. *https://docs.expo.io/introduction/managed-vs-bare/. 2021*

[25] Expo Limitations. *https://docs.expo.io/introduction/why-not-expo/. 2021*