Scaling Real-time Collaborative Editing in a Cloud-based Web App

MASTER THESIS

Martin Dürsch

Submitted on 19 April 2023

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Julian Lehrhuber, M.Sc.
Julia Mucha, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that I have written this thesis unaided and without using sources other than those listed and that this thesis has never been submitted to another examination authority and accepted as part of an examination achievement, neither in this form nor in a similar form. All content that was taken from a third party either verbatim or in substance has been acknowledged as such. The submitted electronic version of the thesis matches the printed version.

Erlangen, 19 April 2023

License

Erlangen, 19 April 2023

Abstract

QDAcity is a cloud-based web application for multiple researchers to collaboratively conduct Qualitative Data Analysis (QDA) in shared projects. Qualitative data (in textual, visual, or audio form) can be open for interpretation and drawing of subjective conclusions. Thus, enabling and encouraging close collaboration can be highly beneficial for teams conducting QDA, since it prevents miscommunication problems. However, real-time collaborative text editing has been missing so far in QDAcity's feature set. In the context of this master thesis, QDAcity's existing collaboration features were extended by real-time collaborative editing of rich text documents. The implementation is required to be compatible with QDAcity's existing features, reliable, maintainable, and extendable. In order to ensure future scalability, the implementation of real-time collaborative editing shall also be horizontally scalable. After discussing various approaches for real-time collaborative editing as well as the scaling of applications in a cloud-based environment, a suitable combination of approaches was chosen, designed, and implemented. The agile development process resulted in the implementation, evaluation, and discussion of Conflict-free Replicated Data Type (CRDT)-based real-time collaborative editing for QDAcity, by harnessing a combination of open-source technologies. Horizontal scalability was achieved by implementing the collaborative editing service in a stateless manner and interconnecting multiple instances via Redis Pub/Sub.

Contents

List of Figures

List of Tables

Acronyms

2PL Two-Phase Locking

ACID Atomicity, Consistency, Isolation, Durability

AWS Amazon Web Services

CAQDAS Computer-assisted qualitative data analysis software

CES Collaborative Editing Service

CRDT Conflict-free Replicated Data Type

DS Differential Synchronization

EC2 Elastic Compute Cloud

FP Functional Programming

FR Functional Requirements

GCP Google Cloud Platform

GCS Google Cloud Storage

HPC High Performance Computing

IaaS Infrastructure as a Service

IT Information Technology

JSON JavaScript Object Notation

LWW Last-Write-Wins

NFR Nonfunctional Requirements

OT Operational Transformation

P2P Peer-to-Peer

PaaS Platform as a Service

QDA Qualitative Data Analysis

RTCS Real-Time Collaboration Service

SaaS Software as a Service

SLA Service-Level-Agreement

TCP Transmission Control Protocol

TDF The Document Foundation

VM Virtual Machine

XP Extreme Programming

1 Introduction

1.1 Motivation

Qualitative Data Analysis (QDA) is a significant part of conducting research in fields that deal with large amounts of qualitative (i.e. non-numerical) data, like psychology, education, pharmaceuticals, and/or other social sciences. Unlike quantitative data, which consists of structured and precise, numerical data, qualitative data can be characterized as unstructured, natural language data in textual, visual, or audio form (Mihas, 2019). Qualitative data usually originates from interviews, documentary materials, field notes, observations, or similar (Graue, 2015). In short, quantitative data deals with numbers, whereas qualitative data deals with meanings (Dey, 2005). QDA can be defined as conducting research by continual reflection about qualitative data, as well as making interpretations, and deriving theories from it (Creswell & Creswell, 2018). It is an established method of theory building in research. An exemplary approach of QDA could be an iterative, alternating process of data analysis and deriving theory from the previously analyzed data (Bryman & Bell, 2011).

Unlike quantitative data analysis, which is accompanied by a lot of statistical and mathematical rules, "[...] there are few well-established and widely accepted rules for the analysis of qualitative data.", according to Bryman and Bell (2011). However, the process of QDA typically involves applying a certain kind of structure to the qualitative data, called "coding". Coding has become a key part of QDA.

Coding loosely describes the process of structuring qualitative data in some way, for instance by assigning specific parts of one or multiple documents to certain "codes". These codes label reoccurring patterns or themes. During the process of QDA, related codes are grouped into a category, and related categories are grouped into more encompassing themes that "[...] describe the data in a form which summarises it, yet retains the richness, depth, and context of the original data" according to Seers (2012). Thus, the codes form a tree-like, structured code system, increasing in abstraction from the leaves to the root.

Applications that enable digitally conducting QDA are grouped as Computer-assisted qualitative data analysis software. One representative of Computer-assisted qualitative data analysis software (CAQDAS) is called QDAcity[1]. QDAcity is a cloud-based web application, developed and operated by the Professorship for Open Source Software at the Friedrich-Alexander-Universität Erlangen-Nürnberg. QDAcity provides an environment for multiple analysts or researchers to collaboratively conduct QDA. Since QDA deals with big amounts of fuzzy and subjective data, enabling researchers to share and discuss different interpretations, ideas, and conclusions can be very beneficial for the process of QDA. The approach of enhancing a process by promoting close collaboration and "shrinking the feedback loop" can also be found in other fields. Agile approaches of software development like Extreme Programming (Beck, 2000) serve as examples of this. However, currently QDAcity only allows the simultaneous collaboration of multiple researchers in a shared project, but not on a more granular level in a shared document.

Real-time collaborative editing of a shared document is a classic form of digitally enabled, close collaboration. This particularly applies when considering that prototypical real-time collaborative editing has been demonstrated as early as 1968 (Akhtar, 2022). Its technological significance has grown since then. Due to the large number of use cases that can benefit from real-time collaborative editing, it has developed to be a desired feature for many applications and products. Given the potential benefits of close collaboration for conducting QDA, enhancing the process of QDA via real-time collaborative editing seems promising. Thus, allowing multiple researchers to simultaneously collaborate, not just in a shared project, but also in a shared document, has become apparent to be a great fit for QDAcity.

1.2 Objective

In the context of this thesis, QDAcity shall be extended to enable real-time collaborative editing of text documents. The main part of the implementation shall be an extension of the existing Real-Time Collaboration Service (RTCS). The frontend and backend of QDAcity shall also be extended to integrate the new functionality. The new feature shall be implemented in a way that combines performance, horizontal scalability, reliability and maintainability while at the same time being implemented and deployed using the existing Google Cloud Platform (GCP) infrastructure and services. The implementation should include acceptance test coverage and be extendable to support collaborative editing of different document types in the future. Chapter 3 provides a more detailed list of requirements.

[1] qdacity.com

The requirements of this thesis mainly describe the functionality that the new implementation shall provide, while also expressing clear preferences for the services and tools that are to be used, primarily those that are already in use. In contrast to this, the requirements are agnostic about the technical details of the implementation and do not specify a technical approach for achieving the expected results. Thus, in order to dynamically adapt to gained insights based on the continuous evaluation of existing technologies, we will take an agile approach.

1.3 Thesis Structure

As a consequence of the technical nature of the feature to be implemented, the structure of this thesis is based on the general structure of an engineering thesis.

Chapter 1 motivates this thesis, sets its objective, and describes its structure. Afterwards, chapter 2 discusses and summarizes the most relevant related work and research for this thesis. This chapter deals with QDAcity's coding editor, real-time collaborative editing as well as scaling a cloud-based web application. Chapter 3 formulates the requirements for the implementation part of this thesis. Subsequently, chapter 4 describes and discusses the architectural modifications that were applied to QDAcity in the context of this thesis. It also discusses the design drafts that were in consideration during the agile development process of this thesis. Chapter 5 discusses and describes the used libraries and QDAcity-specific implementation details of the new real-time collaborative editing feature. Chapter 6 revisits and evaluates the fulfillment of the requirements as stated in chapter 3. Chapter 7 discusses considerations about adaptations to QDAcity that were considered but decided against. Additionally, it presents general considerations about the development process and proposes possible future work. Finally, chapter 8 summarizes and concludes this thesis.

2 Related Work

This chapter summarizes and discusses related work and research that is relevant to this thesis. First, section 2.1 provides a basic overview of QDAcity and its coding editor. In section 2.2, the difficulties and approaches of solving real-time collaborative editing are discussed. Lastly, section 2.3 discusses the topic of scaling a web application, like QDAcity that is deployed in a cloud-based environment.

2.1 QDAcity

QDAcity is a cloud-based web application, developed and operated by the Professorship for Open Source Software at the Friedrich-Alexander-Universität Erlangen-Nürnberg. As mentioned in section 1.1, it is a representative of CAQDAS and provides an environment for multiple analysts or researchers to collaboratively conduct QDA. It is implemented using Javascript/Typescript and React in the frontend and Java and Google Cloud Endpoints in the backend. Additionally, the RTCS handles some live awareness features using Javascript. QDAcity is based on various services provided by GCP, such as Google App Engine[1], Google Cloud Run[2], Google Cloud Storage[3], Google Cloud Datastore[4], Google BigQuery[5], Google Cloud Endpoints[6], Google Speech-to-Text[7], as well as Redis[8].

QDAcity's multitude of features includes:

- A coding editor where most of the QDA is conducted.
- Tools to analyze, customize and add additional data and metadata to codes and codings.

[1] cloud.google.com/appengine
[2] cloud.google.com/run
[3] cloud.google.com/storage
[4] cloud.google.com/datastore
[5] cloud.google.com/bigquery
[6] cloud.google.com/endpoints
[7] cloud.google.com/speech-to-text
[8] redis.com/cloud-partners/google/

- Tutorials that introduce QDAcity's features and an FAQ.
- A project dashboard that displays statistical information.
- Speech-to-text transcription for audio data.
- A coding recommendation system.
- Import and export of documents in various formats like RTF and PDF.
- A project revision history.
- Role-based access control for users of shared projects.
- To-do lists.

For this thesis, the central part of QDAcity is the coding editor, as seen in figure 2.1.

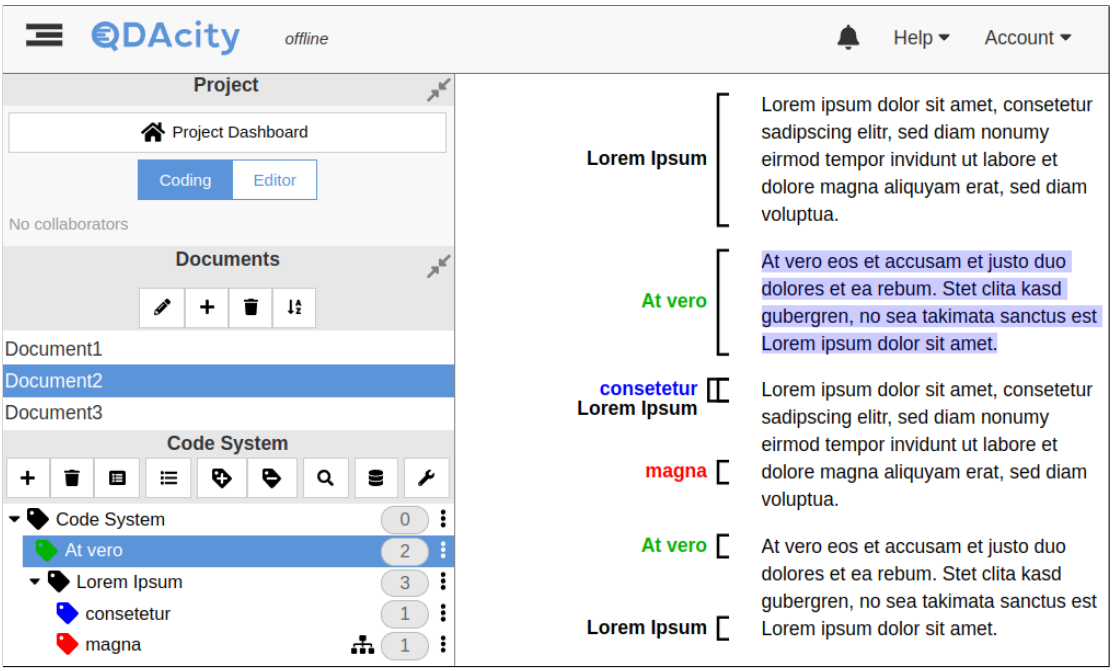


Figure 2.1: Screenshot of a dummy project in QDAcity's coding editor

The two main components of the coding editor are the sidebar and the content area. Within QDAcity, users can create shared projects. A shared project may contain multiple user-uploaded or -created documents in varying formats that share a common code system. The code system is a tree-like structure of codes. The sidebar of the coding editor, on the left side of figure 2.1, gives an overview of these elements. The sidebar also contains most of the buttons for controlling the application, applying codings, and using various tools. Additionally, the sidebar displays if other users are currently working on the same project.

For documents that are represented in an editable format, the content area displays an included and tightly integrated text editor. It is displayed on the right side of figure 2.1. The text editor enables modification of the document and provides rich text features. The left side of the text editor shows coding brackets of codes applied to different parts of the text. Since these can be customized, they are represented in various colors.

2.2 Real-time Collaborative Editing

This section discusses real-time collaborative editing. In the context of this thesis, we have to choose an approach for implementing real-time collaborative editing for QDAcity. Subsection 2.2.1 starts with traditional collaborative editing, like a shared document file. Subsections 2.2.2 and 2.2.3 emphasize the constraints and difficulties that distinguish real-time collaborative editing from traditional collaborative editing. Subsequently, we will discuss how the three most popular approaches to real-time collaborative editing handle these constraints and difficulties. The approaches covered are Differential Synchronization (DS) in subsection 2.2.4, Operational Transformation (OT) in subsection 2.2.5, and Conflict-free Replicated Data Types (CRDTs) in subsection 2.2.6.

2.2.1 Collaborative Editing

In the context of this thesis, we follow the definitions of The Document Foundation (TDF)[9] regarding collaborative editing, real-time collaborative editing, and offline collaborative editing[10]. TDF is a charitable foundation under German law and the developer and maintainer of LibreOffice.

Collaborative editing can be described as multiple users concurrently making changes or applying edits (used synonymously in this thesis) on a document in some form. If a document is represented by a generally indivisible block of binary data (like a Microsoft Word document for example), this will lead to multiple users making changes based on the same version of the document, overwriting each other's changes upon persisting the document. This problem is a type of write conflict called lost updates. In general, a write conflict describes a situation where multiple users (or other kinds of entities) concurrently make equally valid changes to a document (or some other data object), based on the same version, leading to data loss (like with lost updates) or inconsistent results.

[9] www.documentfoundation.org

[10] https://wiki.documentfoundation.org/index.php?title=Collaborative_Editing&oldid=277901

In software engineering, this is a typical use case for concurrency control. There are two main approaches to concurrency control. These are pessimistic concurrency control and optimistic concurrency control (Sheikhan & Ahmadluei, 2013).

The pessimistic approach works by locking a shared document so that only one user may edit the document at a certain time. According to Fraser (2009b), "[...] a familiar [real-world] example is Microsoft Word's behavior when opening a document on a networked drive. The first user to open the document has global write access, while all others have read-only access."

The optimistic approach works without locks. After reading and editing a document, optimistic concurrency control would check whether the document has experienced other changes in the meantime, before overwriting the file. If not, the overwriting operation succeeds. If yes, then the change operation has to be aborted and restarted based on the updated version of the document.

Both of these approaches of concurrency control have in common that they prevent the possibility of write conflicts. Pessimistic concurrency control prevents the possibility of concurrent writes in general, by locking a shared document while it is being opened with write access. Optimistic concurrency control blocks write conflicts by checking for their occurrence before writing a change, aborting the write operation if it would lead to a write conflict. Both of these approaches lead to changes on a document only being possible in serializable order, based on the document's most up-to-date version. This brings the version history of the document into a global total order. A total order means that for any two edit operations, we can always say which one happened first (Kleppmann, 2017). The resulting version graph is a unidirectional path graph (like a single git branch) as can be seen in figure 2.2. The nodes in figure 2.2 represent subsequent versions of the document, edited by two different authors that are called "Green" and "Blue". The edges in figure 2.2 represent the application of edits to convert an old version of the document into a newer version.

Figure 2.2: Document edited by different authors, document versions are in a global total order

Transferring the demonstrated example to a simple web application scenario, the Microsoft Word document would be some kind of document data in a database. The users trying to open and edit the Microsoft Word document would correspond to users on a browser-based frontend. The frontend contains a document editor component that is used to display and edit the document. Figure 2.3 demonstrates this scenario.

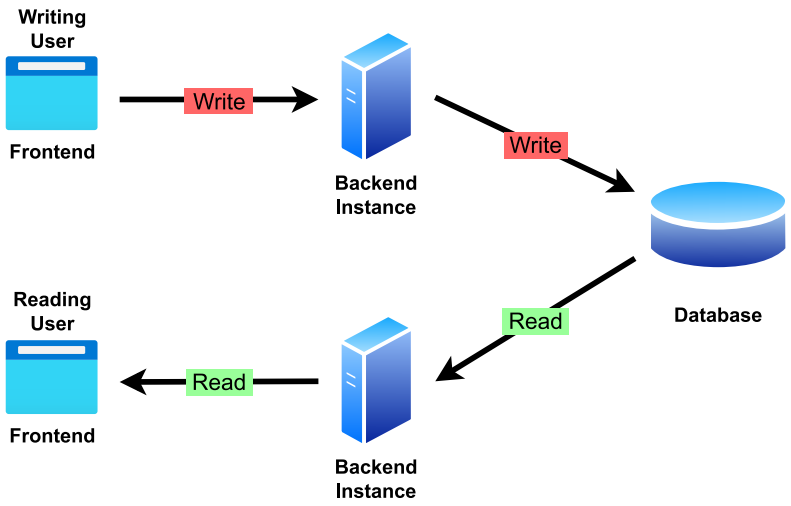


Figure 2.3: Examplary architecture of a simple web application (including write/read path)

According to Kleppmann (2017), "[...] in [the] typical web application model, the database acts as a kind of mutable shared variable that can be accessed synchronously over the network. The application can read and update the variable, and the database takes care of making it durable, providing some concurrency control and fault tolerance." In the context of the web application example, we assume that the database is a single-node ACID-database that uses strong isolation like via Two-Phase Locking (2PL) or serializable snapshot isolation to execute transactions in serializable order (Kleppmann, 2017).

However, multiple users first requesting a document to the frontend and then writing a new version of the document via separate database transactions would again lead to users overwriting each other's changes. To prevent the possibility of lost updates and to be consistent with the Microsoft Word example, we would assume that reading the current version of a document and overwriting it is bundled into a single transaction. Changes on a document are always based on the last written version and are applied to the document in total order. Thus, the database also uses some form of pessimistic or optimistic concurrency control to provide guarantees about the correctness of transactions and prevent write conflicts on the document.

In practice, however, when a user reads the state of a document via the traditional HTTP request/response pattern, he receives a snapshot of the document from a single point in time. The browser assumes that the local state of the document is static unless the user himself locally applies edits. The browser does not subscribe to updates from the backend/database, thus the document is possibly stale before even being displayed in the browser. The only way of finding out whether the displayed data is stale would be to repeatedly refresh the web application in

the browser to check for newer versions. However, repeatedly sending reads via HTTP request will still display snapshots of the document from different points in time (Kleppmann, 2017), which may already be stale on arrival. In computer science, this is referred to as polling. The web browser that displays the snapshot of the data after requesting it can be seen as a temporary, possibly stale cache of the data from the database.

Real-time collaborative editing can be described as multiple users simultaneously editing the same document. In the context of this thesis, we understand real-time collaborative editing as being a seamless experience. Multiple users editing a document, using real-time collaborative editing, shall feel the same way to them as editing the document alone on their local client. The only difference is the following: if multiple users are concurrently editing the same document, the users see each other's changes pop up in real time. In the context of this thesis, real-time is defined as as fast as reasonably possible, given the usual constraints of a distributed system. Furthermore, there is offline collaborative editing. Offline collaborative editing describes a situation where users can keep editing a shared document after disconnecting. While disconnected, users that are concurrently editing a shared document cannot exchange updates. However, these updates will be exchanged once the connection is restored, converging the shared document.

Enabling real-time collaborative editing in a web environment requires solving the two problems of enabling real-time reads and real-time writes. We will discuss these in the following, starting with enabling real-time reads.

2.2.2 Real-Time Reads

A user writing some data and another user reading it can be thought of as a data flow connecting the two users. The arrows between the different domains in figure 2.3 represent this data flow. It can be divided into two different areas. The part of the data flow that is triggered by the writing user is called the write path whereas the part of the data flow that is triggered by the reading user is called the read path (Kleppmann, 2017). Both of these are marked as such in figure 2.3. It is important to emphasize that the write path and read path are not just determined by the travel of data between the different domains of the application, but more importantly, by the general work done that is triggered by the writer or reader. However, in this case, the data travel distance serves as a way of visually representing this "work done".

In order to enable faster reads we need to extend the write path and shorten the read path. For example, a web application like Twitter might prebuild every user's feed page on writes to be able to serve quicker reads (Kleppmann, 2017). Without a prebuilt user feed, answering a user's read request for his feed would require going through the tweets of all his followed users to collect all of their

relevant posts and build the feed page from scratch. This results in extensive querying of the database for lots of data from different users.

With a user's feed page being prebuilt on writes, a read request for a user's feed page can be immediately answered using the prebuilt feed page. Consequently, since work has been shifted from the read path to the write path, the read path for feed pages has been shortened while the write path of new tweets has been extended.

However, rebuilding the user feed from scratch on every relevant write is still an expensive operation. Especially when considering that many prebuilt versions of the user's feed page might never be requested by the user before the feed page is rebuilt again on the next relevant write. In order to avoid fully rebuilding a user's feed page from scratch on every relevant write, it can instead be maintained and updated on relevant writes. In order to keep the prebuilt user feed up to date, relevant writes are required to be pushed to the prebuilt user feed.

This can be seen in figure 2.4. In order to visually represent the extended write path and shortened read path, we assume that the prebuilt user feeds are cached close to the backend instances. Since the prebuilt user feeds are updated on relevant writes to be consistent with the state of the application in the database, they can be seen as an extension of the state of the database. The state of a prebuilt user feed in the cache is derived from the state of the database, which acts as the source of truth (Kleppmann, 2017).

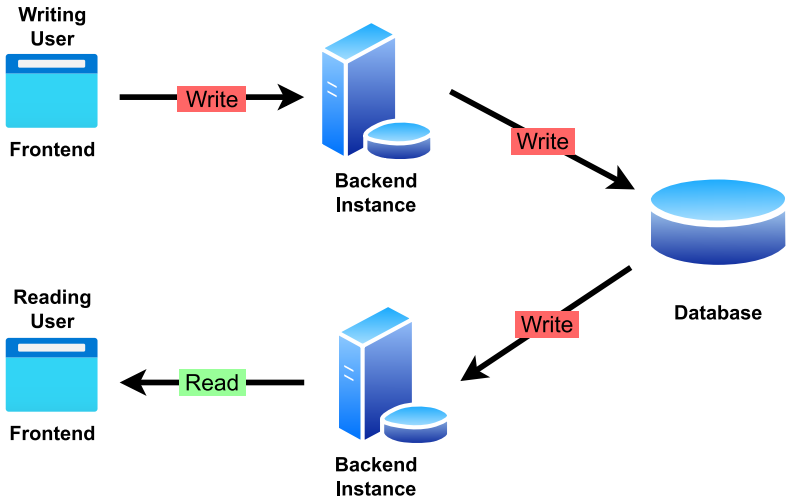


Figure 2.4: Example of an adjusted write/read path for a web application

However, in order to enable real-time reads for real-time collaborative editing, it is not enough to just shorten the read path. Enabling real-time reads requires pushing writes to the readers as soon as they happen, thus extending the write path all the way from the writer to the reader. This results in eliminating the

read path entirely. Instead of waiting for the user to request a snapshot version of the document, changes to the document need to be pushed directly to a user's browser, keeping the document displayed in the browser up to date. This is analogous to the prebuilt user feeds being kept up to date in the cache at the backend instances of figure 2.4. As long as the document is opened in the browser of the user, the state of the document in the user's browser is seen as an extension of the database, which needs to be kept consistent with the state of the database.

This requires moving beyond the basic request/response pattern, to using bidirectional communication protocols like the WebSocket protocol. WebSockets keep an open TCP connection to a server, and the server can actively push messages to the client as long as it remains connected. This enables the backend to actively inform the frontend client about any changes to the state it has stored locally. Upon connection, the client would still request the initial state of the document, involving a read path. Afterwards, the client can rely on the stream of changes sent by the backend (Kleppmann, 2017).

2.2.3 Real-Time Writes

Enabling real-time writes is a more difficult problem. For an application in a web environment like in figure 2.3, traditional collaborative editing using concurrency control and strong consistency would mean the following: in figure 2.3, the database is responsible for preventing write conflicts of a single document by enforcing linearizable writes, and thus a total order of changes. All writes have to be based on the most up-to-date version of the document. Thus, after every write the frontend requests the backend to confirm that the changes made on the current local version of the document have been successful. In this context successful means that the changes did not collide with other changes made in the meantime. If the backend confirms it, the new version will replace the current version in the database. If the backend declines, the local changes will be undone and the document will be refreshed to display the newest version. If this request happens synchronously, the user in the frontend cannot make any further changes until the last change has been confirmed.

In a real-time collaborative editing scenario, in order for a user to be confident that his local version is in sync with the global version of the document, a user would have to pause and wait for confirmation after every keystroke. Furthermore, multiple users editing the same document would result in frequent write conflicts and thus invalidations of local changes. With more users editing the same document, the probability of global write conflicts and thus locally invalidated changes increases exponentially. Of course, one could mitigate this problem by more granularly locking only parts of the document. For example, a database might use locks on a single row level (Kleppmann, 2017) and a document might use one lock per paragraph or line. However, this does not solve the underlying

problem. To make things worse, offline collaborative editing is not possible in this scenario, since the database has to linearize and confirm every edit. To summarize, traditional concurrency control would severely restrict the usability of the editor and not provide a satisfying real-time collaborative editing experience.

At this point, the real-time collaborative editing web application can be described as a distributed system with multiple writing nodes, concurrently writing to the same data in form of the document. In distributed systems theory, allowing only a single writing node (either via concurrency control or as an elected leader node) is not the only way of dealing with the possibility of write conflicts. Instead, one could also allow multiple nodes to write conflicting versions and then handle write conflicts retrospectively. Retrospectively handling write conflicts is a responsibility that can be left to the clients that are using the system or to the system itself. In case of the clients having the responsibility, it could look like this: The distributed data system would be persisting all conflicting versions in a so-called multi-value register (Vitillo, 2022). Then, upon read request, the system would return all conflicting versions to the client. The client would then be in charge of merging the conflicting versions or choosing the most valid one. However, for our use case, a solution that does not hand over this responsibility to the client appears to be more suitable. Another way of dealing with write conflicts would be having the system retrospectively resolve conflicts algorithmically, by merging or deciding on a preferred version (Kleppmann, 2017).

A simple way of handling write conflicts is Last-Write-Wins (LWW), available in many different database systems (Kleppmann & Beresford, 2017). However, as the name suggests, LWW handles write conflicts by accepting the last write (by timestamp) and discarding all other conflicting versions. Thus not a lot is gained from using it for real-time collaborative editing. In order to enable real-time collaborative editing, a more sophisticated conflict resolution algorithm is required.

In order to enable real-time writes, in the sense of real-time collaborative editing, the user needs to be able to make lasting changes in his local client, based on the version that is currently in his local client. In other words, when a user makes changes to his local document, its state must be allowed to temporarily diverge from the state of the document in the rest of the system. Instead of asking the backend/database to approve and write some changes, the backend/database needs to accept the concurrent edits/conflicting versions of multiple users. It is then the responsibility of the backend and the rest of the system to retrospectively resolve write conflicts by converging the state of the document while incorporating all received edits/conflicting versions. The user has to be able to rely on the guarantee that the changes he made will as soon as possible be incorporated into a future version and not get dropped, even when one or multiple other users make changes at the same time.

In summary, the state of the document must be allowed to temporarily locally diverge, yet guaranteed to eventually globally converge. This means a departure from linearizability/strong consistency and a global total order of changes, towards strong eventual consistency, a global partial order, and locally diverging total orders (Alexei Baboulevitch, 2018). Strong eventual consistency is a variation of eventual consistency that guarantees strong convergence. Strong convergence means that a local replica that receives concurrent edits from other replicas, can immediately apply incoming edits to its local version, and still eventually converge. With strong convergence, all incoming edits can be applied to the local state immediately upon arrival, no matter their order of origin or arrival. Replicas that have received and applied the same edits (or the same subset of edits), are guaranteed to have the same local state, no matter the order of application of these edits (Vitillo, 2022).

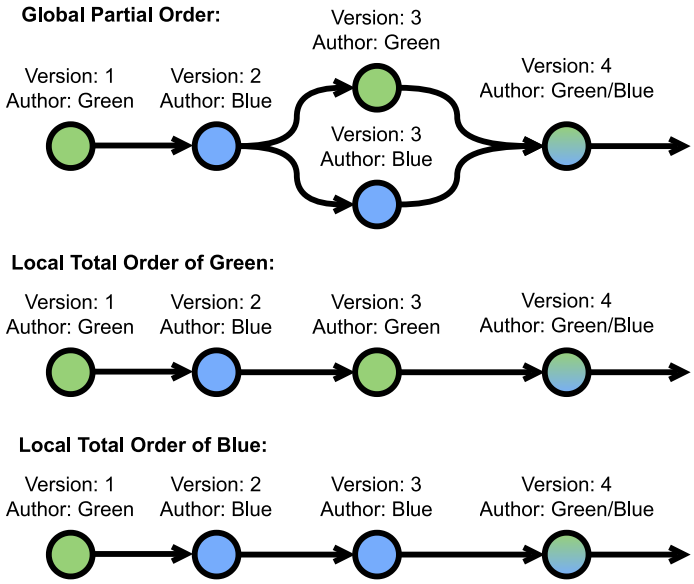


Figure 2.5: Document simultaneously edited by different authors, document versions in global partial order and local total order

Partial order means that, unlike with a total order, pairs of operations exist for which you can not decide which happened first (Kleppmann, 2017). These are the edits that happened concurrently at different clients. Figure 2.5 shows some exemplary version graphs. From a global perspective, the version graph of the document is partially ordered. This means that the version graph may include some changes that happened sequentially as well as other changes that happened concurrently, leading to branching in the version graph. In distributed systems theory, global partial order is usually used in accordance with causal order and causal consistency. (Kleppmann, 2017). In order to keep track of causal order in a distributed system, logical clocks (Lamport, 1978) like Lamport timestamps or vector clocks are used. These work similarly to the version counters in figure 2.5.

Locally, users write and receive changes to their version of the document at different points in time, leading to temporary divergence of the local versions from the other versions of the document in the rest of the system. However, for real-time collaborative editing, all the local versions shall eventually converge into a single global version.

For distributed systems in general there are many constraints that apply (Van Den Hoogen, 2004). One of these is the certainty that network partitions are inevitable in a distributed system (Gilbert & Lynch, 2012). In the context of a web application that offers real-time collaborative editing, being able to tolerate network partitions means supporting offline collaborative editing. Nonetheless, even without network partitions, messages in a packet-based network can arrive indefinitely delayed (for example due to packet congestions). Correctly handling these cases requires the real-time collaborative editing web application to be able to merge an undefined number of concurrent changes, based on different versions of the document, arriving at the same time.

The three most popular approaches to solving real-time collaborative editing are DS, OT, and CRDTs. These will be discussed in the following subsections.

2.2.4 Differential Synchronization

The first solution for real-time collaborative editing we are looking at is DS. DS is an algorithm that has been developed by Neil Fraser at Google (Fraser, 2009b). DS can be described as a further development of three-way merges.

Three-way merges have been used in early forms of collaborative editing (Lindholm, 2004). Nowadays, three-way merges are best known from version control software like git and svn. A three-way merge is what the "git merge" command does when both of the branches that are to be merged, experienced changes since diverging. A three-way merge solves write conflicts, by comparing both of the changed versions with their shared base versions to build a combined version (using a patch function). An example of this is demonstrated in figure 2.6.

Three-way merge uses diff, match, and patch functions to achieve this. How well a three-way merge solves write conflicts depends on the individual implementation of these functions. On the one hand, this means that you can use a three-way merge for basically any use case as long as you have fitting diff-match-patch implementations for your use case. On the other hand, this way, the responsibility for the correct handling of write conflicts is ultimately still on the developer that provides the diff-match-patch implementation. For example, git merge uses a line-based approach to merge changes to the same file (de la Vega & Kolovos, 2022). However, in case of overlapping or overly intermixed changes, it relies on the user to converge conflicting versions into a consistent state.

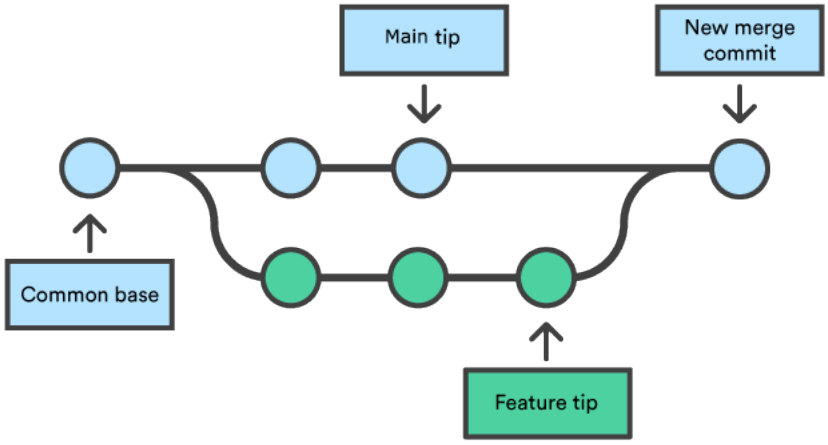


Figure 2.6: Example of a three-way merge using git merge
(Source: https://www.atlassian.com/git/tutorials/using-branches/git-merge)

In 2006, Google implemented its version of diff-match-patch for plain text (also line-based) to power Google Docs[11].

The main problem with three-way merges for real-time collaborative editing is the following: if another user makes any changes to the merging versions while the merge is still in progress, the merged version is basically obsolete before even being finished. Thus, the three-way merge is not useful while users are still actively editing. Think about another git user committing a change to one of the two branches that are to be merged, while the merge is in progress. If this happens, you need to redo the three-way merge from scratch, incorporating the new commit, in order to get an up-to-date result. Fraser (2009b) calls this "[...] the chickens [need to] stop moving so we can count them".

DS is an algorithm that gets around this problem by using an infinite cycle of background diff and patch operations (Fraser, 2009b). In this example, the match step from diff-match-patch is part of the patch operation, resulting in diff-patch. An overview of DS in a local environment can be seen in figure 2.7. In a local environment, the client text and the server text share the same common shadow, whereas, in a network environment, each client/server text has its own shadow. However, the working principle remains the same.

[11] github.com/google/diff-match-patch

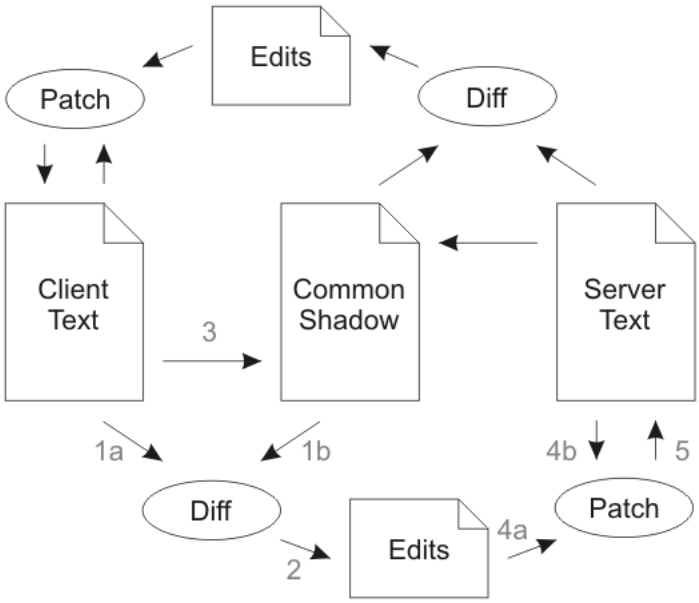


Figure 2.7: Differential Synchronization in a local environment (Fraser, 2009b, p. 2)

According to Fraser (2009a), the algorithm consists of the following steps:

1. First, the Client Text is diffed against the Common Shadow.
2. A list of edits that have been performed on the Client Text is returned.
3. Common Shadow is overwritten with the current snapshot of the Client Text.
4. The list of edits from step two are applied to the Server Text on a best-effort basis.
5. Server Text is updated with the result of previous step.
6. The cycle restarts with the server text at step one.

The enabling mechanism is that the patch algorithm is fuzzy (Fraser, 2009a). This means that the patches are likely able to be successfully applied per line of text to their target text, even if the line changes in the meantime. If a patch can not be successfully applied to a line anymore (due to severe changes of this line in the meantime), another synchronization attempt will be included in the next loop going in the other direction. This leads to repeated attempts at converging, possibly cycling until it succeeds.

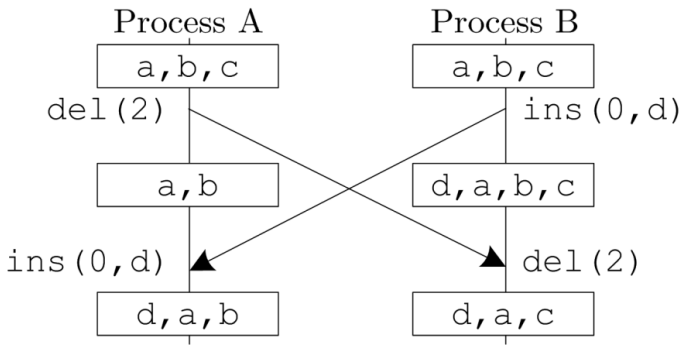
However, eventual automatic convergence is not guaranteed. Patches can not be guaranteed to arrive (in time or at all) in a network environment with best-effort delivery. This might lead to client text and server text (and their local shadows) diverging to an extent that is beyond synchronizable with simple patch operations (Fraser, 2009a). This is because at some point the patch algorithm might not recognize the right lines to apply the line-based edits to anymore. In this case, the process requires human intervention or the two parties have to sync by one side transmitting the whole body of the text to the other side, possibly discarding changes by doing so.

To summarize, with appropriate implementations of diff and patch, DS can basically be applied to enable real-time collaboration in any use case. However, this is a double-edged sword. Similar to the three-way merge, the diff and patch algorithms, which are doing the heavy lifting for write conflict resolution and convergence, are to be provided by the developer. One could describe this as concealing the true complexity of handling write conflicts for real-time collaborative editing behind the abstractions of unspecified diff and patch functions (Alexei Baboulevitch, 2018). Additionally, a developer that is implementing custom diff and patch algorithms for his custom use case will have a hard time mathematically proving that his implementation delivers correct results. To summarize, the patch algorithm is generally fuzzy and works on a best-effort basis.

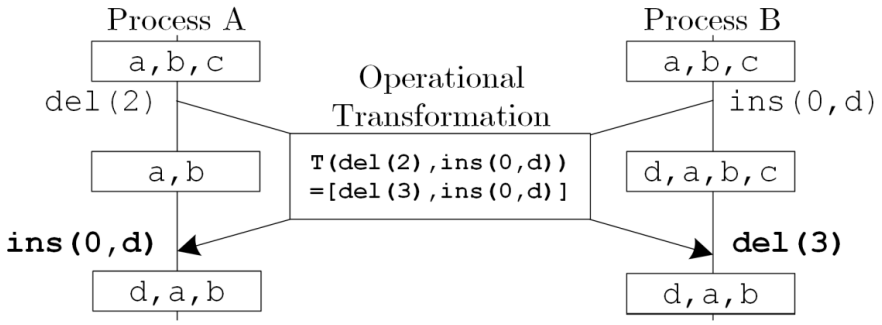
On the one hand, this is advantageous. Whereas other text-based solutions usually work with indexes (like OT for example), DS tries to find the most probable positions for its changes by looking at the surrounding characters (the "match"-part of diff-match-patch). Thus, generally staying true to the intention of the users changes. On the other hand, since it does not rely on a mathematically proven model for convergence (Alexei Baboulevitch, 2018), it is unclear whether it will actually be able to converge the state of two documents. If it cannot, one side must transmit the whole body of the text to the other side to get back in sync. This results in the loss of applied changes. To avoid the sudden loss of data it could stop and ask the users to manually merge the two documents, resulting in a suboptimal user experience. As of 2010, Google replaced three-way merging and DS for Google Docs in favor of OT (Day-Richter, 2010).

2.2.5 Operational Transformation

OT is the most popular solution for real-time collaborative editing, notably currently used in Google Docs (Day-Richter, 2010), Etherpad (‘Etherpad’, 2011, March 26/2011), Dropbox Paper (Sun et al., 2018) as well as many other commercial products. OT resolves write conflicts by transforming concurrent operations. Figure 2.8 shows an example of this.



(a) Inconsistent results.



(b) Consistent with Operational Transformation.

Figure 2.8: Example of OT (Boelmann et al., 2016, p. 4)

Part (a) of figure 2.8 shows two clients making concurrent changes to their local copy of the shared string "abc". The edit operations consist of an operation type (like insert or delete), an index, and possibly one or multiple characters. Transmitting edit operations over the network to other clients takes an unspecified amount of time. This is not a problem as long as edit operations happen non-overlapping in global total order. However, concurrent edits will make the local copies diverge. In figure 2.8, the shared string "abc" locally diverged to "ab" and "dabc". After receiving and executing the edit operations of the other client, the local strings are "dab" and "dac". By simply exchanging concurrent index-based edit operations, the local copies of the shared string have not converged again.

Inserting a character at any position in a string increments the index of all subsequent characters. The edit operations in this format rely on an index to specify the position of the operation. Executing the edit operations in different orders leads to different outcomes. Consequently, edit operations that rely on indexes are not commutative. In this case, whereas del(2) deleted the character c for process A, it deleted the character b for process B due to the changed index of character c after the ins(0,d) operation of Process B.

Part (b) of figure 2.8 shows how OT solves this problem by transforming concurrent edit operations in a way that leads to convergence of the shared string. OT recognizes operations that happen concurrently with other operations based on a version counter/logical clock. Then, if necessary, it adapts the edit operations to the individual local version timelines before applying them. In part (b) of figure 2.8 you can see that the del(2) operation was adapted to the local version timeline of process B by changing it to del(3). However, for the ins(0,d) operation at process A it was not necessary to do so, because this edit operation is not affected by the previous del(2) operation.

OT accepts that in a distributed system, edit operations that are distributed over the network will lead to noncommutative operations being applied in different local orders. It then adjusts these edit operations accordingly to account for these different local edit histories and has the local versions globally converge anyway.

Since OT intercepts and, if needed, modifies edit operations it can be described as an input-based algorithmic solution. Due to the reliance on index-based edit operations, OT is especially suited for real-time collaborative editing of an ordered list of items (Kleppmann, 2017). Thus, OT is not all too complex to implement for plain text editing. However, for rich text editing, due to the sheer number of possibilities of at least two different operations overlapping, it is generally known as being very complex to implement correctly. For example, when adding a new custom edit operation, all the possible interactions with every other existing operation need to be correctly handled (Alexei Baboulevitch, 2018). Due to its complexity, it tends to be more in use by larger companies.

After its first proposal by Ellis and Gibbs (1989), many different versions of OT have been proposed, yet proven to not guarantee convergence (Kleppmann, 2018). For most of them, some edge cases were eventually discovered that were found to produce different results on different clients. With OT this is especially problematic. Because it is an input-based algorithm, once documents go out of sync due to incorrectly handled inputs, they will remain out of sync for at least the remaining session.

Most of the still assumed to be correct versions of OT require a central server that handles each client individually, thus avoiding the complexity of transformations that involve multiple users at the same time. These are the algorithms proposed by Nichols et al. (1995) and Vidot et al. (2000). An assumed to be correct Peer-to-Peer (P2P) implementation of OT was proposed by Oster et al. (2006). This approach works by, among other things, retaining tombstones for removed elements. However, since this approach turned out to be even more complex (Herron, 2020), most applications that use OT rely on a central server.

To summarize, the strengths of OT include the following:

- Since OT works by intercepting and modifying different edit operations, the underlying document data structure itself remains unchanged.
- The intent of a user's change can be represented by the type of operation.
- OT is generally real-world proven when using a central server, due to the number of products that successfully use it.
- Simple implementation for plain text editing.
- OT can support offline collaborative editing for clients by locally buffering operations and then reconciliate with the server upon reconnection (Gentle, 2016).

Nonetheless, there are also some downsides:

- While the client has been offline, many concurrent changes might have happened that need to be taken into account when transforming the locally buffered operations. This leads to potentially expensive operations upon reconnection.
- OT is complex to implement for rich text editing from the ground up, due to the large number of cases and edge cases that need to be considered.
- Since OT only works on the inputs, a single faulty transformation is enough for documents to be permanently out of sync.
- Not well suited for P2P-based real-time collaborative editing.
- Since a single central server is responsible for algorithmically combining all concurrent edit operations, the performance of OT can suffer dramatically with high numbers of concurrent edit operations (Li & Li, 2006) or clients recovering from network partitions.

2.2.6 Conflict-Free Replicated Data Types

The third and youngest approach to real-time collaborative editing is based on CRDTs. According to Kleppmann (2017), CRDTs "[...] are a family of data structures for sets, maps, ordered lists, counters, etc. that can be concurrently edited by multiple users, and which automatically resolve conflicts in sensible ways".

Although CRDTs have been around for longer, they have first been formalized as a category of data structures by Shapiro et al. (2011). They are mathematically sound, based on commutativity and monotonic semilattices. Thus, provided a communication network that ensures eventual delivery, "CRDTs are guaranteed to [eventually] converge towards a common, correct state, without requiring any

synchronization", according to Shapiro et al. (2011). Since then, more complex data structures like nested JSON objects have been implemented as CRDTs (Kleppmann & Beresford, 2017), enabling more complex use cases like real-time collaborative, rich text editing.

According to Bourgon (2014), "the tl;dr on CRDTs is that by constraining your operations to only those which are associative, commutative, and idempotent, you sidestep a lot of the complexity in distributed programming." These three properties are what different CRDTs have in common:

- Commutativity: Commutativity is defined as the property that the order of applied operations does not change the outcome, i.e. f(g(x)) = g(f(x)). Well-known commutative operations are addition (a + b = b + a) and multiplication (a * b = b * a).
- Associativity: Associativity is defined as the property that the grouping of operations does not change the outcome. Again, well-known associative operations are addition (a + (b + c) = (a + b) + c)) and multiplication (a * (b * c) = (a * b) * c).
- Idempotence: Idempotence is defined as the property of an operation that, when performed only once has the same effect as when performed multiple times. Setting a key in a key-value store to some fixed value is an idempotent operation, since writing the value again simply overwrites the value with an identical value. However, incrementing a counter is not idempotent since performing the increment again means the value is incremented twice (Kleppmann, 2017).

These properties are achieved by attaching additional metadata to the data structure and its contents (Kleppmann & Beresford, 2017). This additional metadata could be IDs for all elements, tombstones for deleted elements, or logical/vector clocks.

Figure 2.9 shows an example of a CRDT for real-time collaborative plain text editing. Both replicas (p and q) start with the string "abc". Both replicas make some changes. Then their local edits are exchanged via the network. It is important to note that the indexes that are used as parameters for the edit operations are only used as an API to point to characters in the text. Internally, unique IDs are used to access certain characters. This is contrary to the approach of OT from the previous subsection 2.2.5. OT tries to identify characters in a text document by index position and then algorithmically fixes synchronization problems of distant clients by transforming index-based operations. CRDTs avoid the problem of shifting indexes by using the additional metadata to uniquely identify all characters by ID.

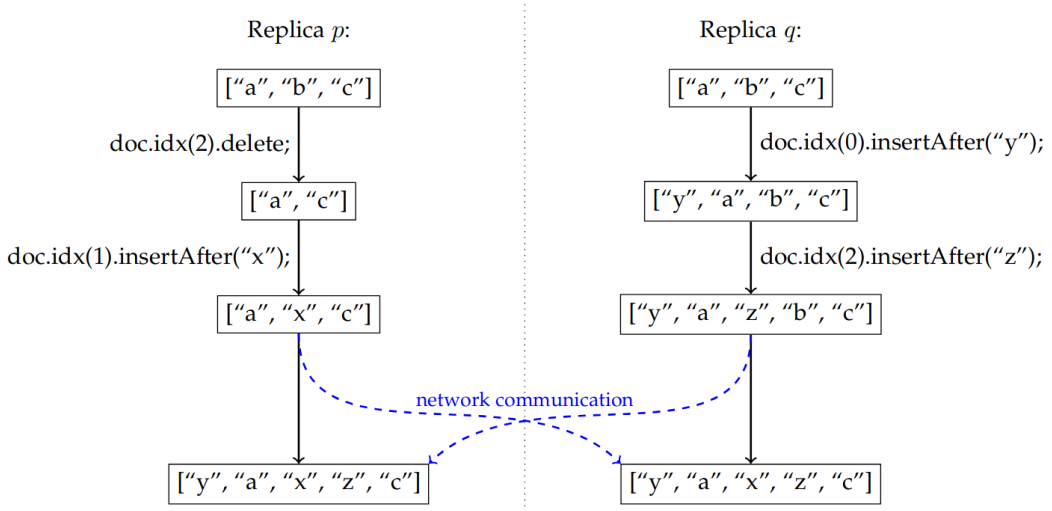


Figure 2.9: Concurrent editing of plain text using a CRDT (Kleppmann & Beresford, 2017, p. 5)

Replica p deletes the 'b' character using an index as a parameter. Since replica q added a 'y' character at the beginning of the string and inserted a 'z' character in front of 'b', the index of subsequent characters got incremented. This includes the 'b' character, whose index was incremented by two. However, since CRDTs rely on unique ids to access single elements, the shifted indexes do not prevent the delete operation at replica q from finding the right character to delete.

Both replicas add one new character, 'x' and 'z', behind 'a'. Because these operations happen concurrently, after exchanging the edits it is unclear whether 'x' or 'z' is supposed to be first in order. In this case, the CRDT uses an unambiguous secondary sorting property that ensures that the result of concurrent operations will be the same on every replica. Thus, the replicas are guaranteed to converge.

The listed properties enable CRDTs to provide eventual consistency without OT-like transformations, without concurrency control (Sun et al., 2018) and even without a central server. In a P2P environment or leaderless data system, every participating replica might broadcast received edit operations to all other participating replicas. This would be a problem for a simple OT algorithm, due to the nonidempotent operations. However, the idempotence property of CRDT operations ensures that this is not an issue. For example, a P2P environment might lead to a unique delete operation arriving multiple times at a replica. In this case, a tombstone in the metadata of the CRDT signals that the repeatedly received delete operation has already been executed and can be ignored.

In general, CRDTs can be described as approaching the problem on a lower level than OT, by building data structures that are inherently tolerant of the constraints of distributed systems. As claimed by 'Peter Bourgon on CRDTs, Go at SoundCloud' (2015), "the key win for CRDTs is that [as a developer] you get to ignore all the problems that are inherent in distributed systems, so network failures, message duplication, out of order arrival [...]".

However, there is a significant downside of using CRDTs. Because this is a data structure-based approach, developers are forced to map the data (and thus logic) that is supposed to converge to appropriate CRDTs. Since OT works on the edit operations, real-time collaborative editing can be implemented without touching the underlying data structures. Implementing CRDT-based real-time collaborative editing to an existing application is a more extensive operation.

In practice, CRDTs are deployed for many use cases beyond collaborative text editing. CRDTs are especially prevalent as registers, counters, maps, and sets for so-called Dynamo-style key-value stores like Dynamo, Cassandra, or Riak (Vitillo, 2022). For example, Riaks CRDTs have been part of the implementation of the chat feature of a multiplayer game that handled 70 million unique monthly players at the time (Ptaszek, 2014). Beyond that, CRDTs are in use by many tech companies like Meta (Shi et al., 2020), Apple (Hedkvist, 2021) and Soundcloud (Bourgon, 2014).

To summarize, using CRDTs for real-time collaborative editing includes the following strong points:

- CRDTs' defining properties allow developers to avoid dealing with lots of constraints of distributed systems, reducing development complexity.
- CRDTs are naturally convergent. No algorithmic conflict resolution mechanisms are required, further reducing complexity.
- Due to CRDTs' defining properties being inherent to the data structure itself, no central server is required. This enables P2P environments, high scalability, and high availability.
- CRDTs' defining properties enable offline-mode support with almost no overhead.

On the other hand, weaknesses of CRDTs for real-time collaborative editing include:

- CRDTs reliance on additional metadata, may affect long-term performance. The reliance on tombstones for deleted elements, for example, leads to monotonically increasing document size as more operations are applied, even when content is deleted. This also makes CRDTs more suitable for manually edited documents than automatically written documents.

- Whereas OT defines and works on different edit operations that are close to the actual user operations, CRDTs' state-transforming operations are more low-level. Consequently, these operations do not capture the user's intent as accurately. This may lead to users having to intervene if a shared document converges in an unexpected way.
- CRDTs are a data structure-based approach to real-time collaborative editing. Consequently, adding real-time collaborative editing to existing applications requires translating the existing underlying data structures and the logic that acts upon them to CRDTs. This requires major modifications.

2.3 Scaling a Cloud-Based Web Application

This section presents the topic of scaling a web application in a cloud-based environment. After providing a general overview of cloud computing as an environment for scaling applications in subsection 2.3.1, we will discuss the two basic categories of scaling applications. These are vertical scaling, as described in subsection 2.3.2, and horizontal scaling, as described in subsection 2.3.3 (Dutta et al., 2012). Furthermore, we will outline the topic of scaling dynamically based on fluctuating workloads, called elasticity, in subsection 2.3.4.

According to Sotiriadis et al. (2019), scalability refers to the ability of the system to accommodate larger loads (mid to long-term adjustments), whereas elasticity refers to the ability to adapt to loads dynamically (short-term adjustments). Scaling an application could include taking measures in terms of its underlying hardware resources as well as adapting an application's software and structure in order to better deal with the increasing load. When scaling an application, there are different kinds of load whose scaling must be approached in different ways. For example, scaling the amount of data an application can hold differs from scaling the volume of requests an application can process (Kleppmann, 2017). Elasticity is enabled in particular by on-demand resource provisioning that public cloud platforms provide.

2.3.1 Cloud Computing

The emergence of cloud computing had a big impact on facilitating the provisioning of underlying hardware resources to an application. According to Armbrust et al. (2010), the term cloud computing refers to both the cloud itself (the hardware and software in the data centers) as well as the services provided by those data centers. These are usually divided into Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). A cloud that is made available to the general public is called a public cloud while an organization's internal cloud is called a private cloud. The public clouds with the most

market share as of 2023 are (in descending order) Amazon Web Services (AWS), Microsoft Azure and GCP. According to Dutta et al. (2012), the emergence of public clouds was enabled primarily by advances in server virtualization (for instance containerization) and has been a major transformation in the IT industry in the last 15 years.

According to Armbrust et al. (2010), on the public cloud provider side, the key advantages of cloud computing compared to traditional medium-sized, private data centers are economies of scale and multiplexing. Due to economies of scale, building extremely large-scale, commodity hardware data centers at optimal locations vastly reduces the costs of hardware, software, maintenance, and operation. This applies in particular when comparing large-scale public cloud data centers to smaller, on-premise data centers. Hardware is expensive, not just to purchase but also to operate. The cost of operation of a server does not scale down linearly with its utilization. This is because an idling server will still draw a lot of power and needs to be maintained. Consequently, higher utilization of existing hardware in a data center is generally more efficient and preferred. As stated by Armbrust et al. (2010), back in 2008, "[...] real world estimates of average server utilization in data centers range from 5% to 20%. This may sound shockingly low, but it is consistent with the observation that for many services the peak workload exceeds the average by factors of 2 to 10. Since few users deliberately provision for less than the expected peak, resources are idle at nonpeak times. The more pronounced the variation, the more the waste." Making large-scale data centers available to the general public greatly improves the utilization rate of existing hardware, since a variety of workloads of different customers balance each other out and lead to smaller fluctuations in utilization. In consequence, while a small-scale, company private, on-premise data center may experience big fluctuations between low and high utilization, a large-scale public cloud data center is generally able to balance its utilization on a higher, more efficient level.

On the customer side, using services provided by a public cloud instead of building and operating a small to medium-sized private data center includes the following benefits (Armbrust et al., 2010):

- Less financial risk: The elimination of a big up-front financial commitment of building a data center, including acquisition and operation of hardware, enabling small startups with low resources or big risk-averse companies to innovate using data center services.
- Easy scaling on demand: The ability to provision what appears to be infinite computing resources on demand, thus enabling an application to scale alongside possibly rapidly growing demand without having to rely on long-term growth predictions to acquire a sufficient amount of hardware resources beforehand.

- Elasticity: The ability to automatically adjust to fluctuations in workload and only pay for services that are actually being used, financially rewarding the freeing of resources that are no longer needed.

Since the objective of this thesis includes implementing a scalable service in a cost-efficient way, these benefits are significant to us. Even if a cloud computing provider might set a price for using a server per timespan that is higher than the cost of operating the same server for the same timespan by yourself, the operational and economical benefits of easier scaling and elasticity as well as less financial commitment usually outweigh the potentially lower costs of self-hosting.

2.3.2 Vertical Scaling

As mentioned before, vertical scaling can be defined as adding more resources to existing application instances. As stated by Pujol et al. (2010), "a natural and traditional solution to cope with higher demand is to upgrade existing hardware". This includes upgrading the capabilities of the CPU, memory, storage, network bandwidth, etc. With specialized hardware, you can join together many CPUs, RAM chips, and disks under one operating system and a fast interconnect allows any CPU to access any part of the memory or disk. This so-called shared-memory architecture can still be treated as a single machine (Kleppmann, 2017).

As stated in subsection 2.3.1, vertical scaling is significantly easier in a cloud computing environment compared to a traditional self-hosted environment. Whereas in a conventional environment, you had to purchase hardware, install it and get it up and running with appropriate drivers, in a cloud computing environment it is usually a matter of just a few clicks to upgrade a Virtual Machine (VM) to stronger hardware or a cloud service to a higher tier. Modern hypervisors might even support adding additional resources to a VM without stopping it (Dutta et al., 2012), a process called hot adding. However, with VM-focused cloud services like Amazon Elastic Compute Cloud (EC2) [12] and Google Compute Engine[13] the execution of the VM usually still needs to be stopped in order to upgrade its instance type. This is because an upgrade of its instance type will most likely trigger a live migration of the stopped VM to a different physical machine (Dutta et al., 2012). This process avoids restarting the VM and re-initializing its applications, thus preserving its state. However, live migration of a VM will still incur some unavailability and delay in executing incoming requests. The time frame of a live migration can take anywhere from a few seconds to several minutes, depending on different circumstances like the size of the VM, availability of resources, or whether the VM shall be migrated to a different region/data center.

[12] docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-resize.html
[13] cloud.google.com/compute/docs/instances/changing-machine-type-of-stopped-instance

Positive aspects of vertical scaling include:

- Vertical scaling of VMs in a cloud-based environment usually only takes a few clicks and some downtime for live migration.
- Vertical scaling is an easy way of increasing an application's processable load, without having to implement complex adaptations of its software.

On the other hand, vertical scaling also has significant drawbacks (Kleppmann, 2017):

- As stated by Kleppmann, "the problem with [vertically scaling a machine using] a shared-memory approach is that the cost grows faster than linearly: a machine with twice as many CPUs, twice as much RAM, and twice as much disk capacity as another typically costs significantly more than twice as much. And due to bottlenecks, a machine twice the size cannot necessarily handle twice the load." Vertically upscaling a single shared-memory machine will quickly lead to a departure from commodity hardware, towards hardware specialized for High Performance Computing (HPC), incurring huge costs. As mentioned in subsection 2.3.1, using commodity hardware in high volumes is much more price efficient and a key characteristic of cloud computing.
- A vertically scaled, special hardware machine might be able to process the same load as multiple commodity hardware machines. However, unlike multiple commodity hardware machines, it is still a single point of failure. This generally leads to lower availability.
- Related to the previous point, a vertically scaled, shared-memory architecture system is definitely limited to a single geographic location. This leads to high latencies from distant regions.
- In many cases, workloads have reached a size that makes vertical scaling technically unfeasible. For example, already by the years 2008-2010, Facebook (now Meta) required "multiple hundreds of Terabytes of memory across thousands of machines", according to Pujol et al. (2010).

Due to these significant drawbacks, scaling an application nowadays usually refers to horizontal scaling.

2.3.3 Horizontal Scaling

Instead of adding more resources to existing application instances, horizontal scaling can be defined as adding additional application instances to the cluster or instance group. Since these instances are executed separately, this is known as shared-nothing architecture (Kleppmann, 2017). In a cloud computing environment, instances can refer to physical instances like a VM running the application,

or logical instances, like serverless functions or an application process that is running in a VM. A physical instance can usually host multiple logical instances. In the context of this thesis application instance refers to a logical instance.

The complexity of horizontally scaling an application can differ significantly, depending on whether the part of the application that should be horizontally scaled is stateless or stateful. In the context of this thesis, we will use definitions of statelessness and statefulness based on those formulated by Red Hat[14]. A stateless application is defined as an application whose application instances can understand and handle requests in isolation and do not locally store or require information on past requests. Therefore, an application instance can be replaced interchangeably with any other identical application instance. Any user request can be routed to and processed by any application instance (Kleppmann, 2017). A stateful application, however, is defined as an application whose application instances execute requests using the context of previous requests. Stateful requests can be thought of as ongoing conversations with the same person. In order to have a valid dialogue with a person, consecutive requests need to be answered by the same person. Consequently, an instance of a stateful application cannot be replaced interchangeably with other instances of the same application since their states might differ.

Nowadays, web applications are usually deployed as stateless applications. As has been said in subsection 2.2.1, the database in a traditional web application schema acts as a passive, mutable, shared state that provides concurrency control and fault tolerance. The application instances, on the other hand, act as stateless functions that mutate the state of the database. This separation of state and function is also generally known as a fundamental aspect of Functional Programming (FP).

The separation of state and function makes stateless applications easy to scale horizontally. Since instances of stateless applications can respond to requests interchangeably, horizontally scaling a stateless application may only require adding additional instances behind a load balancer. Application instances of stateless applications can be removed and added at will, without interfering with other running application instances. As claimed by Pujol et al. (2010), "horizontal scaling has eased most of the scaling problems faced by traditional web applications. Since the application front-end and logic are stateless, it can be instantiated on new servers on demand in order to meet the current load." The functional aspect of stateless applications is also reflected in the names of some popular cloud services that offer serverless hosting of stateless applications and automatic horizontal scaling. These include Azure Functions[15] and AWS Lambda[16].

[14] www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless
[15] learn.microsoft.com/en-us/azure/azure-functions/
[16] aws.amazon.com/lambda/

Horizontally scaling stateful applications is far more complex since it generally requires carefully designed logic, and/or partitioning of the application state. Partitioning can be defined as splitting up the state of an application or database into multiple disjoint partitions that are distributed among the set of application instances. Requests that try to query a specific partition of the application state need to reach the specific instance that is responsible for this partition. This requires some kind of service or mechanism that handles the routing of requests to the required application instance. This is an example of a common problem called service discovery (Kleppmann, 2017), further adding complexity to the stateful application.

The complexity of partitioning the state of an application or database varies significantly, depending on the structure of the data that it holds. For example, partitioning the key-value data of a hashmap, by key range or hash of the key, is rather straightforward (Kleppmann, 2017). This is because the data is already strictly separated by its keys. However, partitioning data that is highly interconnected and does not already have clear boundaries is far more complex (Pujol et al., 2010). Data from a social network for example is usually stored in the form of graph databases. A graph database generally consists of vertices (entities) and edges (relationships). Every pair of vertices can potentially be connected via one or multiple edges. On a social network, a user can potentially be friends with every other user but also be connected with all kinds of other entities like locations, posts, groups, events, etc. A graph database is a natural way of modeling this highly interconnected data (Kleppmann, 2017). Partitioning this kind of highly interconnected data is non-trivial, leading to the development of specialized algorithms and procedures (Pujol et al., 2010).

Regardless of whether data is easily partitionable or highly interconnected, partitioning requires careful consideration (Kleppmann, 2017). In order to distribute partitions among all of the application instances, the number of partitions needs to be at minimum the number of application instances. When an application scales horizontally, the number of application instances may increase and at some point even exceed the number of partitions. At this point, in order to redistribute the application state to all the application instances, the whole application state needs to be repartitioned more granularly from scratch. In order to avoid an expensive repartitioning process, it is generally advisable to partition the application state rather granularly from the beginning (assuming the workload is expected to grow). This results in one application instance handling multiple partitions. Once certain partitions become a so-called hot spot, either in terms of their size or in terms of query load, the partitions need to be rebalanced. In this case, only specific application instances may be affected by the rebalancing. A highly granular partitioning process allows for easy allocation of an appropriate volume of partitions (in terms of total size and query load) to a given application instance. However, high granularity comes with a high over-

head of metadata. Thus, when partitioning the application state, it is required to find a balance between the required granularity (for example based on estimated workload growth) and the metadata overhead.

On the one hand, the advantages of horizontal scaling include the following:

- Vast scaling potential, enabling scaling of applications that cannot feasibly be scaled vertically.
- Cloud providers offer services for fast and easy horizontal scaling of stateless application instances.
- Since horizontal scaling is relying on commodity hardware, it is more cost-efficient compared to vertical scaling.
- Generally higher availability, because the application is not relying on a single instance point of failure.
- Instances in different regions possible, leading to low latencies for users.

On the other hand

- Horizontally scaling stateful applications usually requires significant adaptions of software and infrastructure.
- If required, partitioning, service discovery, and rebalancing introduce a lot of additional complexity.
- Horizontal scaling brings up different bottlenecks at different magnitudes of scaling. During the growth process of a horizontally scaled application, its structure needs to be regularly adapted to the increasing load.

In summary, when it comes to horizontal scaling, if possible, it is preferable to keep application instances stateless in order to avoid introducing the additional complexity of scaling stateful applications. It should be mentioned, however, that most applications can not avoid handling some kind of state. If not in the service layer, then usually in an underlying database layer. Keeping application instances in the service layer stateless for easier horizontal scaling may only shift the complexity from the service layer to the underlying database layer. For handling large amounts of data and complex state in the database layer, there are many different existing databases. These may be open source or closed source, relational or nonrelational, provide strong or weak isolation levels, are single node or distributed, etc. Correctly choosing, configuring, and using a horizontally scalable, distributed database brings a lot of challenges and complexity on its own (Kleppmann, 2017) that cannot be ignored, whether by stateful or stateless application instances. However, discussing this topic in detail would exceed the scope of this thesis.

2.3.4 Elasticity

As stated in chapter 2.3.1, many data centers and web applications experience some kind of workload fluctuation, depending on the kind of service they provide. Due to the natural day and night cycle, human-used web applications are especially affected by this. Based on sampling data of a private data center of a Fortune 500 company, Dutta et al. (2012) observed that for half of the intervals, the required short-term scaling factor exceeded 2.

Scaling a web applications infrastructure to handle the expected workload peaks generally leads to overprovisioning, as seen in figure 2.10 (Armbrust et al., 2010).

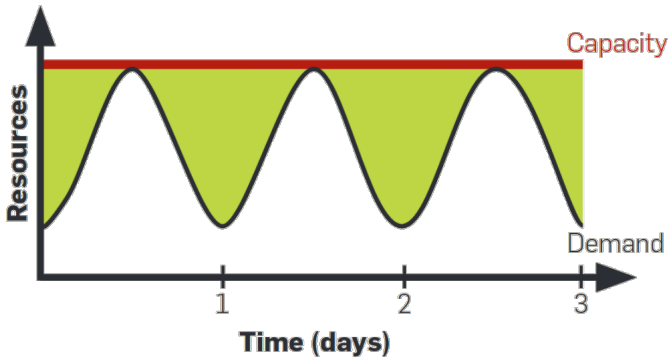


Figure 2.10: Provisioning for peak load (Armbrust et al., 2010, p. 54)

The x-axis in figure 2.10 represents the passage of time in days, while the y-axis shows the capacity provisioned/required by demand. The fluctuating demand is shown in black and the static capacity to handle peak demand is in red. The green area represents the difference in demand and capacity during non-peak times. When overprovisioning, (even if you could perfectly anticipate peak load like in this example) the green areas are wasted capacity, since the application has more resources available than required to fulfill its Service-Level-Agreement (SLA) (Lorido-Botran et al., 2014). Of course, you could try to optimize static capacity for the least difference in capacity and demand, or, maybe more practically applicable, minimization of cost of operation plus estimated cost of SLA violation (Sotiriadis et al., 2019). Regardless of whether it was done deliberately or is a consequence of underestimated load, the resulting state is called underprovisioning and is shown exemplarily in figure 2.11.

However, provisioning less capacity than required to handle peak load leads to service unavailability during peak load. Long-term, this service unavailability may lead to negative press and unsatisfied customers switching to the competition, as represented by figure 2.12.

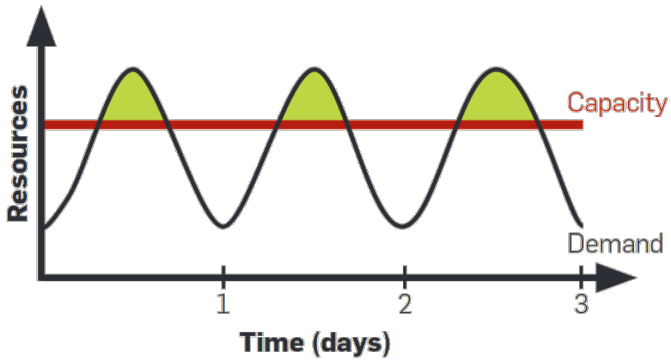


Figure 2.11: short-term underprovisioning (Armbrust et al., 2010, p. 54)

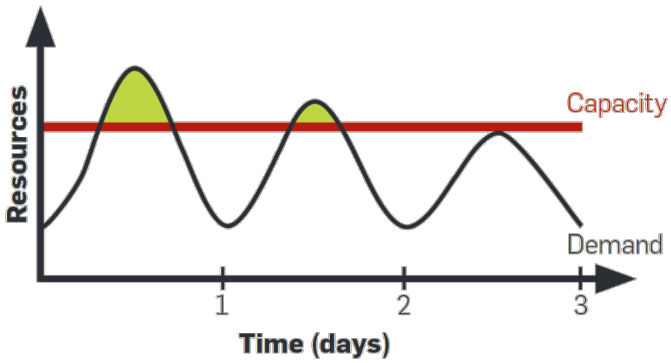


Figure 2.12: long-term underprovisioning (Armbrust et al., 2010, p. 54)

This is where elasticity comes into play. AWS[17] defines elasticity as "[...] the ability to acquire resources as you need them and release resources when you no longer need them. In the cloud, you want to do this automatically." Other definitions of the term elasticity, e.g. as stated by Kleppmann (2017), generally refer to the capability of a system to do this automatically. As mentioned in 2.3.1, one key advantage of using cloud computing compared to private data centers is the ability to granularly add or remove resources within minutes rather than weeks or months. This capability enables customers of public clouds to automatically align their provisioned resources with the experienced demand, almost entirely eliminating the difference between provisioned resources and experienced demand.

In order to achieve this automatically, “[...] an auto-scaler is in charge of making decisions about scaling actions, without the intervention of a human manager. [...] Auto-scaling decisions rely on having useful, updated performance metrics.

[17] wa.aws.amazon.com/wellarchitected/2020-07-02T19-33-23/wat.concept.elasticity.en.html

The performance of the auto-scaler will depend on the quality of the available metrics, the sampling granularity, and the overheads (and costs) of obtaining the metrics", as stated by Lorido-Botran et al. (2014).

Performance metrics that may be taken into account by an auto-scaler include (Lorido-Botran et al., 2014):

- Hardware metrics like CPU utilization, memory usage, and disk access load.
- Operating system metrics like CPU-time per process or percentage of page faults.
- Load balancing metrics like the number of jobs/requests in queues or response times.
- Database metrics like the number of transactions, number of aborted transactions to ensure transaction isolation, and response times.

An auto-scaler may be a simple rule-based agent, some kind of more sophisticated algorithm, as proposed by Kwan et al. (2019) and Dutta et al. (2012), or based on neural networks to predict future workload (Rossi et al., 2019).

Nowadays (as of 2023), many cloud computing services offer features for configuring fully automated horizontal scaling. Usually, high-level PaaS services that leave the management of instances to the cloud provider can be expected to offer such features.

A sophisticated auto-scaler may use a combination of vertical and horizontal scaling measures to adjust provisioned capacity to the experienced load. Whether to choose vertical or horizontal scaling measures depends on different factors, for example, the expected longevity of a certain fluctuation in workload or the ease with which application instances can join or leave the cluster.

Horizontal scaling of stateless applications is generally problem free since new application instances can be added to the cluster without affecting existing instances.

However, horizontal scaling of stateful applications may require a full restart due to reconfiguration or rebalancing of partitions (see subsection 2.3.3) This leads to short-term unavailability potentially affecting every application instance (Dutta et al., 2012). This is an example where a hasty horizontal scaling action could make the situation worse, not better. In this scenario, migration of only a few VMs, triggered by vertical scaling, appears to be the better choice, especially for handling short-term fluctuations (Dutta et al., 2012). In order to keep a stateful application running and available when scaling horizontally, the application needs a (potentially complex) live migration procedure that correctly handles the migration of certain partitions of the application state/workload from one VM/instance to another.

Finding the most cost-effective combination of horizontal and vertical scaling options is difficult. When given a limited amount of physical resources (e.g. VMs) and a set of variable size workloads to distribute among them (like stateful application instances with variable size partitions of application state/workload), finding the cost-optimal configuration is a variation of the bin packing problem, which is known to be NP-hard (Kwan et al., 2019). Consequently, to keep overhead low, an efficient auto-scaler will rely on heuristics instead of computing optimal results.

Regardless of the type of auto-scaler, it needs to avoid "oscillation". Oscillation is an effect that occurs when an auto-scaler executes scaling actions too nervously, leading to an erratic alternation between over- and underprovisioning. Usually, a cooldown period between scaling actions and/or a generously sized utilization target is used to prevent this behavior (Lorido-Botran et al., 2014).

3 Requirements

This chapter states the requirements for the new real-time collaborative text editing feature. The requirements are categorized into Functional Requirements (FR), listed in section 3.1, and Nonfunctional Requirements (NFR), listed in section 3.2. After describing the templates that the requirements are based on, the requirements and their associated sub-requirements are listed. The syntax of the requirements is based on the requirement templates by Rupp and SOPHIST-Gesellschaft für Innovatives Software-Engineering (2014).

3.1 Functional Requirements

The representation of the FR follows the FunctionalMASTeR template by Rupp and SOPHIST-Gesellschaft für Innovatives Software-Engineering (2014), as seen in figure 3.1.

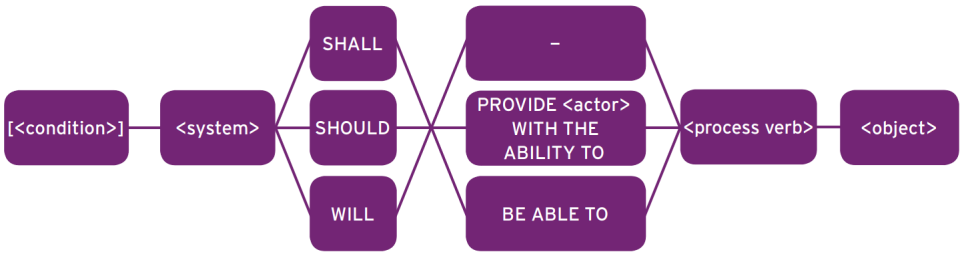


Figure 3.1: FunctionalMASTeR template (Rupp & SOPHIST-Gesellschaft für Innovatives Software-Engineering, 2014, p. 230)

The FunctionalMASTeR template describes the syntax of expressing FRs in structured, natural language. The system that is to be developed forms the grammatical subject of the requirement. In this case, it is the aforementioned Collaborative Editing Service (CES). The keywords "shall", "should" and "will" inform about the degree of obligation to fulfill the respective requirement (in descending order).

The square brackets around the condition in the leftmost box mark this part as optional. The syntax of conditions is based on the ConditionMASTeR template by Rupp and SOPHIST-Gesellschaft für Innovatives Software-Engineering (2014), as seen in figure 3.2.

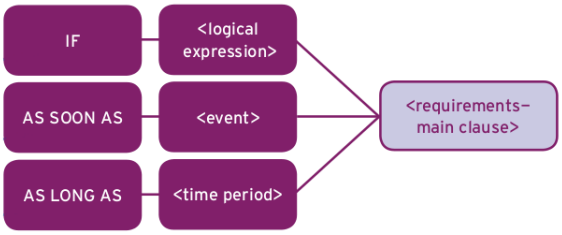


Figure 3.2: ConditionMASTeR template (Rupp & SOPHIST-Gesellschaft für Innovatives Software-Engineering, 2014, p. 242)

The ConditionMASTeR template divides conditions into 3 categories. These consist of conditions based on logical expressions (keyword "If"), conditions based on events (keywords "As soon as"), and conditions based on time periods (keywords "As long as").

FR-1: The RTCS shall be extended by a new CES that generally enables real-time collaborative editing for users of QDAcity.

> **FR-1.1:** In order to keep overhead for maintenance and deployment low, the CES shall be an extension of the QDAcity RTCS.
>
> **FR-1.2:** The existing Slate[1] text editor in the QDAcity frontend shall be extended to connect to the CES and make use of the real-time collaborative editing functionality that the CES provides.
>
> **FR-1.3:** In order to provide a consistent user experience in the coding editor, the CES or reworked RTCS shall be able to keep the code system synchronized with the text document.
>
> **FR-1.4:** The CES shall only process the operations of authenticated and authorized users.

FR-2: The real-time collaborative editing feature that the CES provides shall enable efficient document synchronization across multiple users.

> **FR-2.1:** As soon as a user requests a document that is not currently being edited, the CES shall load this document to memory and provide it to requesting users.

[1] docs.slatejs.org

FR-2.2: As soon as a user edits a document, the CES shall accept and distribute these edits to all other concurrent editors of this document in real-time, by using a bidirectional communication protocol.

FR-2.3: As long as one or multiple users are concurrently editing a document, the CES shall prevent write conflicts by continuously converging the state of the document.

FR-2.4: As long as a document is actively being edited, the CES should regularly persist the current state of the document to the backend.

FR-2.5: As soon as the last editing user of a document has disconnected, the CES should persist the most up-to-date state of the document to the backend.

FR-2.6: As long as a document is not actively being edited, the CES should not hold it in memory.

3.2 Nonfunctional Requirements

NFRs that describe properties of the new feature are formulated based on the PropertyMASTeR template by Rupp and SOPHIST-Gesellschaft für Innovatives Software-Engineering (2014), as shown in figure 3.3.

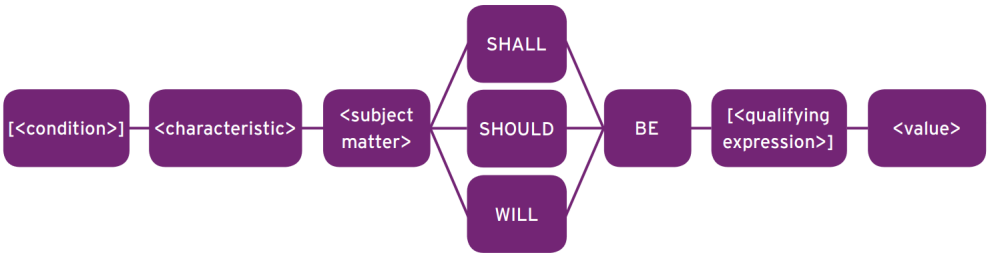


Figure 3.3: PropertyMASTeR template (Rupp & SOPHIST-Gesellschaft für Innovatives Software-Engineering, 2014, p. 239)

NFRs demanded by the operating environment of the CES, are formulated based on the EnvironmentMASTeR template by Rupp and SOPHIST-Gesellschaft für Innovatives Software-Engineering (2014), as shown in figure 3.4.

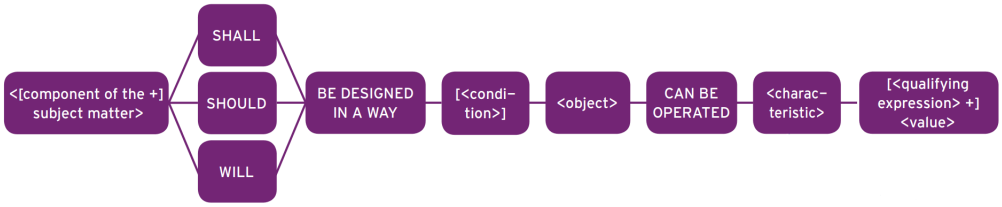


Figure 3.4: EnvironmentMASTeR template (Rupp & SOPHIST-Gesellschaft für Innovatives Software-Engineering, 2014, p. 239)

NFR-1: The CES should be designed in a way that can be deployed and run in the environment that Google Cloud Run provides.

> **NFR-1.1:** Google Cloud Run offers a maximum of 32 gigabytes of main memory per instance.[2]
>
> **NFR-1.2:** Google Cloud Run offers a maximum of 1000 concurrent WebSocket connections per instance.[3]
>
> **NFR-1.3:** Google Cloud Run offers a maximum of 60 minutes per WebSocket connection before timing out.[4]

NFR-2: The CES shall be designed in a way that provides compatibility for existing features and possible feature extensions in the future.

> **NFR-2.1:** The CES shall be implemented in a way that does not break the existing functionality of QDAcity.
>
> **NFR-2.2:** The CES shall be designed in a way that could support syncing additional document formats in the future.
>
> **NFR-2.3:** The CES should be designed in a way that is compatible with offline editing.

NFR-3: The CES should be designed in a way that provides a degree of scalability and elasticity, while not deviating from the GCP-based services that are already in use by QDAcity.

> **NFR-3.1:** The CES shall be designed in a way that is horizontally scalable to cope with increasing load.
>
> **NFR-3.2:** The CES should be designed in a way that can make use of the elasticity that its underlying GCP services provide.

[2] cloud.google.com/run/quotas
[3] cloud.google.com/run/docs/about-concurrency
[4] cloud.google.com/run/docs/triggering/websockets

NFR-4: The CES shall be designed in a way that provides sufficient performance.

> **NFR-4.1:** The CES should be designed in a way that can support the real-time collaborative editing of a single document by at minimum five users.
>
> **NFR-4.2:** The CES should be designed in a way that can support the real-time collaborative editing of at minimum 50 average-size documents on a single CES instance.
>
> **NFR-4.3:** In order to provide a satisfying degree of responsiveness, the average latency of distributing edits between connected users should be less than 1000 milliseconds.

NFR-5: The implementation of the CES should be designed in a way that enables sufficient maintainability.

> **NFR-5.1:** The implementation of the CES should be designed in a way that is well documented internally (commented code) as well as externally (in the QDAcity Wiki).
>
> **NFR-5.2:** The implementation of the CES should be designed in a way that can easily be extended.
>
> **NFR-5.3:** The implementation of the CES should be designed in a way that supports the creation of acceptance Tests.

NFR-6: The CES should be designed in a way that is reliable and fault-resistant.

> **NFR-6.1:** The CES should be designed and deployed in a way that can consistently and autonomously recover from crashes and disconnects.

NFR-7: The CES should be designed in a way that benefits usability.

> **NFR-7.1:** The CES should be designed in a way that supports intuitive understanding by its users by displaying feedback for connection problems.
>
> **NFR-7.2:** The CES should be designed in a way that supports intuitive understanding by its users by displaying text editing awareness information.

4 Architecture

This chapter discusses the architecture of QDAcity, respectively before and after the implementation of the real-time collaborative editing feature. First, the most relevant conclusions for QDAcity from the conducted research are listed in section 4.1. Section 4.2 describes the initial architecture and procedures of QDAcity. Thereupon, section 4.3 describes the reworked architecture and procedures of QDAcity. Section 4.4 discusses different possible data storage formats, since these affected the subsequent architectural decisions. Finally, section 4.5 describes and discusses several approaches to horizontally scaling the new real-time collaborative editing feature of QDAcity. These approaches have been in consideration during the agile development process.

4.1 Research Conclusions

The research that was conducted in the context of section 2.2 has led to the following conclusions for the architecture of the real-time collaborative editing feature for QDAcity:

- In order to enable real-time reads, a bidirectional communication protocol like WebSockets is required.
- In order to enable real-time writes, an approach is required that can guarantee the eventual convergence of simultaneously edited documents.
- DS has been assessed as a bad fit for QDAcity. This is because it can not guarantee convergence without possibly discarding edits. Discarded edits could lead to a bad user experience both for collaborative text editing as well as keeping the code system synchronized. Additionally, a lot of complexity regarding the handling of write conflicts is not actually solved by the approach of DS but by the underlying algorithms DS relies on.
- OT is preferable to DS since it can guarantee convergence. However, the complexity that is inherent to OT as an algorithmic approach could be problematic. Even just extending an existing OT implementation with

additional custom operations grows exponentially in complexity since the interaction of the new operation with all existing operations needs to be accounted for. The real-time collaborative text editing feature for QDAcity should also support synchronizing the code system, additional document formats, and possibly features that are yet to be implemented. Due to the exponentially increasing complexity when adding additional operations to an OT implementation, two separate versions of OT are likely to be required to develop for synchronizing both the text document as well as the code system. Since many other features need to be developed and maintained for QDAcity, an approach with lower complexity and a wider range of applicability would be preferable.

- CRDTs have been assessed as the most promising approach to implementing real-time collaborative text editing for QDAcity. The ability of CRDTs to converge without an explicit synchronization algorithm make it a rather low-complexity approach. The ability to synchronize not just text, but JSON objects in general, enables a wide range of applications for CRDTs. For QDAcity, this capability could be used to synchronize both a text document, as well as the code system of a shared QDAcity project. Additionally, the capability of generally synchronizing JSON objects among multiple users enables extending the system with additional document formats or new collaborative features. The data structure-based approach of CRDTs requires mapping parts of QDAcity's underlying data and business logic to CRDTs. However, this appears to be a more straightforward problem to solve than debugging some kind of faulty OT operation in a distributed system. Consequently, the complexity-reducing properties of CRDTs appear to be a good fit for QDAcity.

In order to provide scalability for the new feature, the following conclusions for QDAcity have been drawn from the research presented in section 2.3:

- The scalability of a vertical scaling approach is inherently limited. Even though horizontal scaling requires additional efforts in terms of software design and architecture, its advantages of providing long-term scalability make it preferable.
- In order to keep the complexity of the scaling reasonably low, a horizontal scaling approach that relies on stateless application instances would be preferred over one that relies on stateful application instances.
- Many cloud computing services offer automated horizontal scaling and elasticity features, usually for stateless application instances. In order to keep the required active maintenance of the new feature moderate, an approach that could be automated using these features would be preferred.

4.2 Initial Architecture of QDAcity

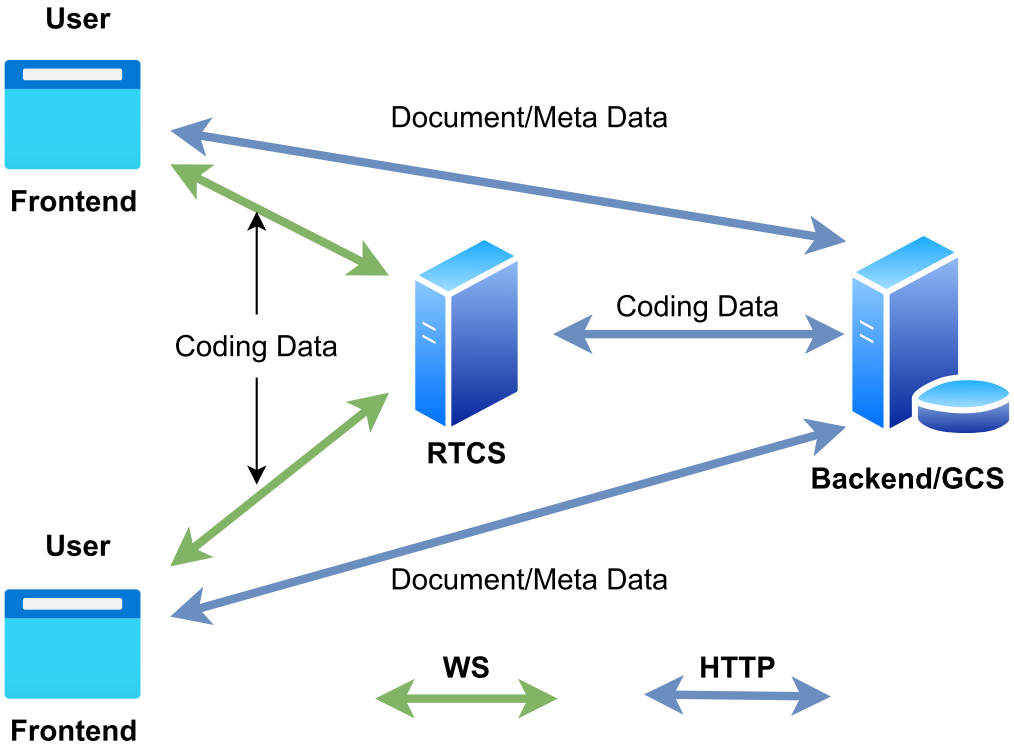


Figure 4.1: Initial architecture of QDAcity

Figure 4.1 shows the initial architecture of QDAcity at the beginning of this thesis. The frontend is based on Javascript using React[1] and styled-components[2]. It is bundled and shipped using Webpack[3]. The RTCS is implemented using Javascript and hosted on Google Cloud Run. The backend is based on Java and uses Google Cloud Endpoints. The backend is responsible for persisting document data in Google Cloud Storage buckets and meta data in Google Cloud Datastore (a NoSQL database).

[1]https://react.dev
[2]https://styled-components.com
[3]https://webpack.js.org/

The initial workflow of editing and coding a document includes the following steps:

1. When opening a project, the frontend requests project data, initial coding data, and other meta data from the backend via HTTP and connects to the RTCS.
2. When opening a document in the project, the frontend requests document data from the backend. The document received from the backend can be seen as a snapshot of the state of the document in the backend.
3. The user edits the local version of the document.
4. The user applies some codes to the document. These get distributed to other users in the same project via the WebSocket connection to the RTCS. Additionally, the RTCS is responsible for forwarding changes in the coding data to the backend via HTTP.
5. The user closes the editor and the local state of the document is persisted by overwriting the state of the document at the backend via HTTP.

While the RTCS is responsible for keeping the coding data in sync, this did not apply to the document itself. Multiple users that would concurrently open, edit and save a document would lead to write conflicts, in particular lost updates. Reading a document and writing changes to the document to the backend did not happen as part of a single transaction (see subsection 2.2.1). Additionally, no sophisticated conflict resolution mechanism was in place that would handle new versions of the document that were based on the same version. Consequently, only the version of the last user saving the document in a state compatible with the coding information at the backend was persisted. The edits of all other users that concurrently edited the same document were discarded.

4.3 Reworked Architecture of QDAcity

In accordance with the research conclusions summarized in section 4.1, the real-time collaborative editing feature has been implemented using CRDTs. The CRDT implementation that was used will be examined and discussed in section 5.1. For this section, we will focus on the architectural details.

As has been stated in the requirements, the CES, which is responsible for keeping concurrently edited documents in sync, is to be implemented as an extension of the RTCS. In order to keep documents and coding data reliably consistent with each other, the old mechanism of distributing coding data was also adapted to CRDTs.

In order to keep the local states of documents in sync from the beginning, users opening a document will receive the state of the document from the RTCS via the WebSocket connection. In order to prevent users from overwriting the persisted document version in the backend with their local and possibly diverging states of the document, only the RTCS will persist documents to the backend. The same principles apply to coding data. The reworked architecture of QDAcity is displayed in figure 4.2.

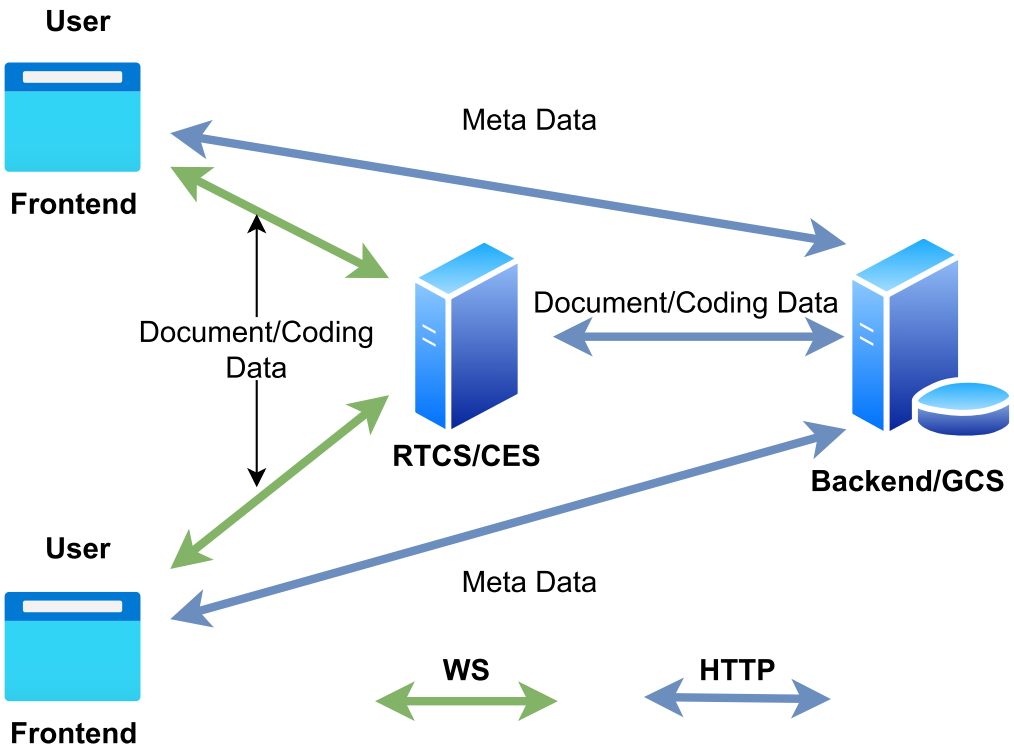


Figure 4.2: Reworked architecture of QDAcity

The reworked workflow of editing and coding a document includes the following steps:

1. When opening a project, the frontend requests project data and other meta data from the backend via HTTP and connects to the RTCS. If there is no open session of the coding data being synced via the RTCS, the RTCS requests the coding data from the backend. The RTCS will deliver the coding data of the project to the frontend via WebSocket. As long as the WebSocket connection exists, the coding data of the project will be kept in sync with all other connected users.
2. When opening a document in the project, the frontend requests the document from the RTCS via WebSocket. If there is no open session of the document being synced via the RTCS, the RTCS requests the document

from the backend. This is necessary because the CRDT implementation and thus the convergence of the document is based on Javascript. The RTCS delivers the document to the frontend. As long as the WebSocket connection exists, the document will be kept in sync with all other connected users.

3. The user edits the local version of the document. The document locally diverges from the rest of the system. The edits are being transmitted to the RTCS. The RTCS applies the incoming changes to its local state and forwards them to all other connected users. The global state of the document converges. The RTCS persists the state of the document to the backend at regular intervals.
4. The user applies some codes to the document. The RTCS handles the convergence of the coding data the same way it handles the convergence of the document.
5. The user closes the document and the frontend closes the WebSocket connection that keeps the document in sync. When the last user closes the document, the RTCS persists the last state of the document to the backend and ends the document session.
6. The user closes the project and the frontend closes the WebSocket connection that keeps the coding data in sync. When the last user closes the project, the RTCS persists the last state of the coding data to the backend and ends the coding data session.

4.4 Document Storage Format

Initially, the state of a document was persisted as a text file containing the document content in the form of HTML. Since the state of the document needs to be present in the format of a CRDT to be synchronized, the question arose in which format the new RTCS should persist documents to the backend. The discussion of options for the persistence format of documents analogously applies to the persistence format of coding data.

The following options were discussed:

- Original HTML format: By using the original HTML format, the least adaptations are required. This particularly applies to the backend. The backend provides features to analyze the existing projects to derive information and provide it to the users. The original HTML format is a good fit because it can be read and, if needed, edited using both Javascript/Typescript in the frontend/RTCS as well as Java in the backend. However, the HTML format has a downside. The RTCS can only apply incoming changes to its

local version of the synced document in the native CRDT form. In this case, when a document editing session ends the RTCS converts the CRDT form of the document to the HTML format and persists it to the backend. Discarding the CRDT form of the document is a hard cutoff point for users that are locally editing a document using offline collaborative editing and are expecting their changes to be retroactively applied to the global state of the document, once they reconnect to the RTCS.

- Native CRDT format: Persisting the document in its native CRDT format would allow the RTCS to retroactively apply edits of users that reconnect after an indefinitely long time period. However, since the used CRDT implementation is Javascript-based, the Java backend likely can not read or edit the document in its native CRDT format without some form of a Javascript runtime environment integration. In order to use the existing analysis code of the backend, the RTCS is required to persist the native CRDT format and additionally derive an HTML version of it for analysis purposes. Furthermore, the native CRDT format has the property of growing monotonically in size with applied operations, regardless of the type of operation (even deletes). This is due to the meta data that is required to guarantee the special properties of the CRDT (see subsection 2.2.6). If the size of the document in native CRDT format grows out of hand, a reinitialization of it would compress it by discarding the built-up meta data. However, this again would be a hard cutoff point for retroactively applying incoming changes.
- Update-based format: The third option is persisting the document as a log of independent and immutable CRDT-updates. Although storing the state of a document and storing the collection of updates of a document might appear to be two different things, they are actually "two sides of the same coin". The most up-to-date snapshot of the mutable state is equal to the aggregation of all updates (Kleppmann, 2017). When initializing an empty document and applying all updates, the output will be the current version of the document. Yet, by converting a sequence of updates to the current state of the document, the information about the temporal progress of updates is lost. Thus, storing the document as a collection of independent updates enables more options for analysis. With a document in its most up-to-date state, you can only perform some specified analysis on this exact version of the document. If you need information about some older version of the document, you can only hope to have performed the analysis of the older version when it was the most up-to-date, or hope to have created a checkpoint of the required document version using QDAcity's built-in document checkpoint feature. With a log of independent updates, every snapshot of the document that ever existed at the RTCS could be rebuilt. This enables retroactively analyzing older document versions with newly

implemented analysis tools. Furthermore, older document versions could be rebuilt and delivered to the frontend for a version history feature that would allow a more granular document history than QDAcity's current document checkpoint feature. The disadvantage of this approach would be the increased storage size of a document at the backend. Applying all updates to build the current state of a document discards all unnecessary information, thus shrinking in size compared to the collection of updates. However, QDAcity's document checkpoint feature also introduces a certain redundancy in terms of disk space usage. In order to keep the required disk space per document under control, one could decide to only store a certain recent time period in the form of independent updates and compress older updates by applying them to a cutoff version of the document. All in all, the estimated value added by an update-based document format over the existing document checkpoint feature was not enough to warrant the implementation effort.

After discussing the mentioned options, their advantages and disadvantages, as well as estimating the expected effort of implementation, we decided to persist the documents in their native CRDT format and add derived HTML versions of the document for analysis purposes.

4.5 Horizontal Scaling Design Drafts

This section discusses design drafts for the implementation of the horizontal scalability for the real-time collaborative editing feature of QDAcity. The design drafts are described and compared using a scenario of four users simultaneously editing two of four persisted documents. First, we will describe the initial state without the implementation of horizontal scaling as a starting point in subsection 4.5.1. Afterwards, three different approaches to horizontal scaling of the RTCS that were in consideration during the agile development process will be described in subsection 4.5.2, subsection 4.5.3, and subsection 4.5.4.

4.5.1 Initial Situation without Horizontal Scaling

Figure 4.3 shows how a single RTCS instance handles the described scenario.

In its current state, the RTCS loads a document into memory as long as it is being edited. This is necessary because the used CRDT implementation is based on Javascript, thus the convergence of the documents can only happen in a Javascript runtime environment.

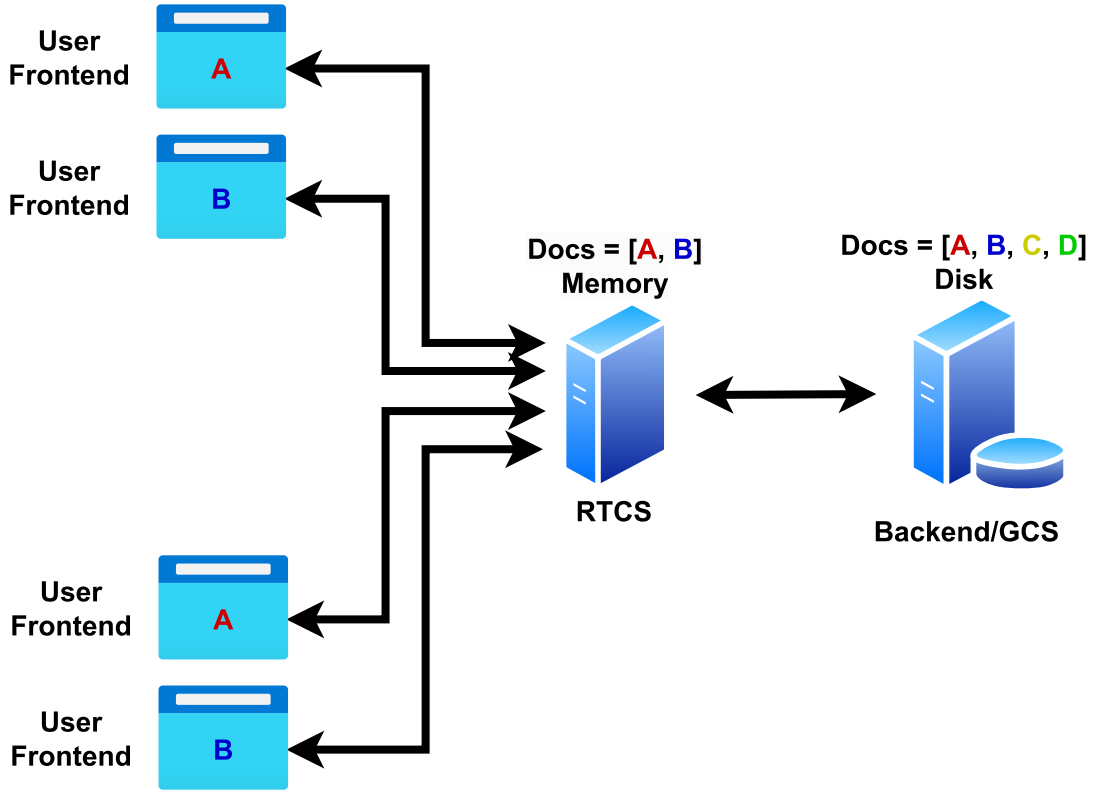


Figure 4.3: Initial situation with a single RTCS instance

Since the RTCS is being deployed on Google Cloud Run, the most likely bottlenecks for a single instance are one of the following:

- The aggregated memory consumption of concurrently edited documents fills up the main memory of the Google Cloud Run instance. At minimum, this is 512 megabytes (mebibytes to be specific, for second generation runtime[4]). For the highest tier, this is 32 gigabytes.
- The number of concurrent WebSocket connections reach Google Cloud Runs maximum of 1000 per instance.

While a single RTCS instance provides great capacity in itself, these constraints represent hard limits in terms of the maximum amount of workload that the RTCS can handle. We will now look at three different approaches to horizontally scaling the RTCS.

[4]https://cloud.google.com/run/docs/configuring/memory-limits

4.5.2 Stateful Instances with Service Discovery/Routing

As mentioned in subsection 2.3.3, in the context of this thesis, stateful instances are defined as instances that hold a partition of the total state of the application. With service discovery/routing, requests that need to act on a certain partition of the state need to be routed to the correct instance. In figure 4.3, the single RTCS instance would load all documents that are currently being edited in memory. With the approach of horizontal scaling via stateful instances, the edited documents are uniformly distributed among the RTCS instances.

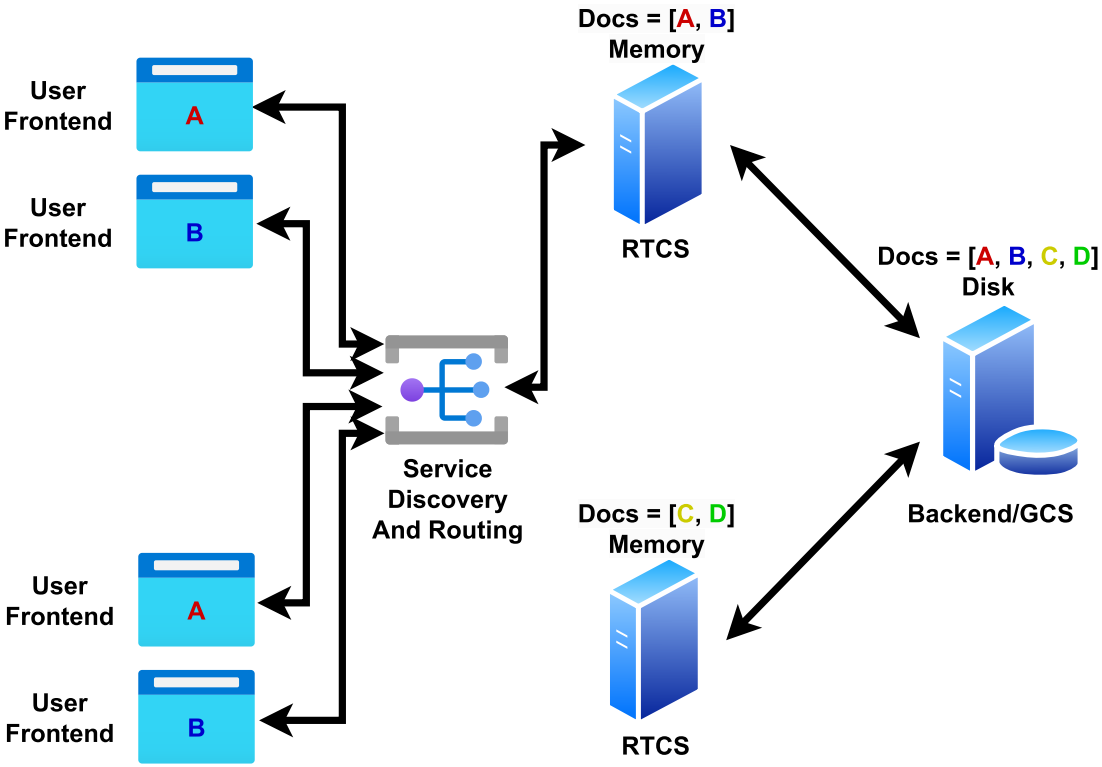


Figure 4.4: Scaling the RTCS as stateful instances with a discovery/routing service

Figure 4.4 shows how horizontally scaling the RTCS as stateful instances would look like. The documents A, B, C, and D have been distributed evenly to the two separate RTCS instances. Since the RTCS instances equally handle a distinct partition of the total application state, this leads to great scaling capabilities. This partitioning of state however requires a middleware that routes the requests to the correct RTCS instances. If we were to scale this approach, by adding or removing some instances, the total state of the application would have to be repartitioned and distributed for the new number of RTCS instances.

Since QDAcity is a user-centric application, large short-term fluctuations in workload are to be expected. For example, due to the day and night cycle. Additionally, the load per document is expected to be rather irregular, since different documents will be hotspots at different times. The latter aspect may balance itself after a threshold number of documents per RTCS instance. However, the efficient handling of the day and night cycle is definitely expected to require short-term elasticity, once a certain size of the user base is reached. Consequently, a mechanism that is able to seamlessly live migrate document editing sessions between RTCS instances without restarting the whole cluster for reconfiguration of partitioning would be required. Additionally, the service discovery and routing middleware would have to be synchronously notified of repartitioning, rebalancing, and live session migrations to provide correct routing at all times.

All in all, while this approach can deliver both, high scalability and elasticity, the complexity overhead of stateful RTCS instances lead us to consider solutions with stateless RTCS instances.

4.5.3 Stateless Instances with Log-Based Message Broker

In order to remove the statefulness of the RTCS instances, it is conceivable to design an approach where the whole state of the application is stored in the Backend and the RTCS instances only serve as functional actors that mutate the state in the backend.

In subsection 4.5.1, it was stated that for the initial architecture of the real-time collaborative editing feature, the documents need to be loaded into the RTCS, because the application of state updates to the CRDT document requires a Javascript runtime environment. In its current configuration, QDAcity can execute Javascript in the frontend and in the RTCS, but not in the Java-based backend.

In section 4.4, it has been said that manifested state is equal to the aggregation of all state changes. For the initial architecture of the real-time collaborative editing feature, the RTCS immediately applies incoming updates to the state of the document and then persists the manifested state of the document to the backend. Instead, one could remove the step of immediately applying state updates and only persist the sequence of incoming document updates to the backend without manifestation of the current state in the RTCS or the backend. When a user wants to access a certain document, the frontend would request the sequence of updates from the RTCS and then build the current state of the document from the sequence of updates in the Javascript-based frontend.

However, in order to keep the manifestation of the state of the document in the frontend up-to-date, it should be possible for the client to subscribe to incoming

changes of documents at the backend. Log-based message brokers like Apache Kafka[5] appear to be a good fit for this use case.

When using Apache Kafka, producers append messages to an immutable log (Kreps et al., 2011). Old and new messages can then be read by registered consumers. Messages that are produced for Apache Kafka are grouped in logical channels called topics. For our use case, all updates of a document are logically connected, thus there would be one topic per document. Since each user should receive all update messages of a document, a topic should not be split up into multiple partitions and each consumer should have its own consumer group.

When users access a document from the frontend, the RTCS will act as a producer for the topic of the document and append the edits of the user. At the same time the RTCS instance will register as a consumer to the topic to receive incoming edits from other users working on the same document. After building the state of the document in the frontend, the frontend will use the WebSocket connection to the RTCS as a proxy to produce and consume edit messages for the particular document topic of Apache Kafka.

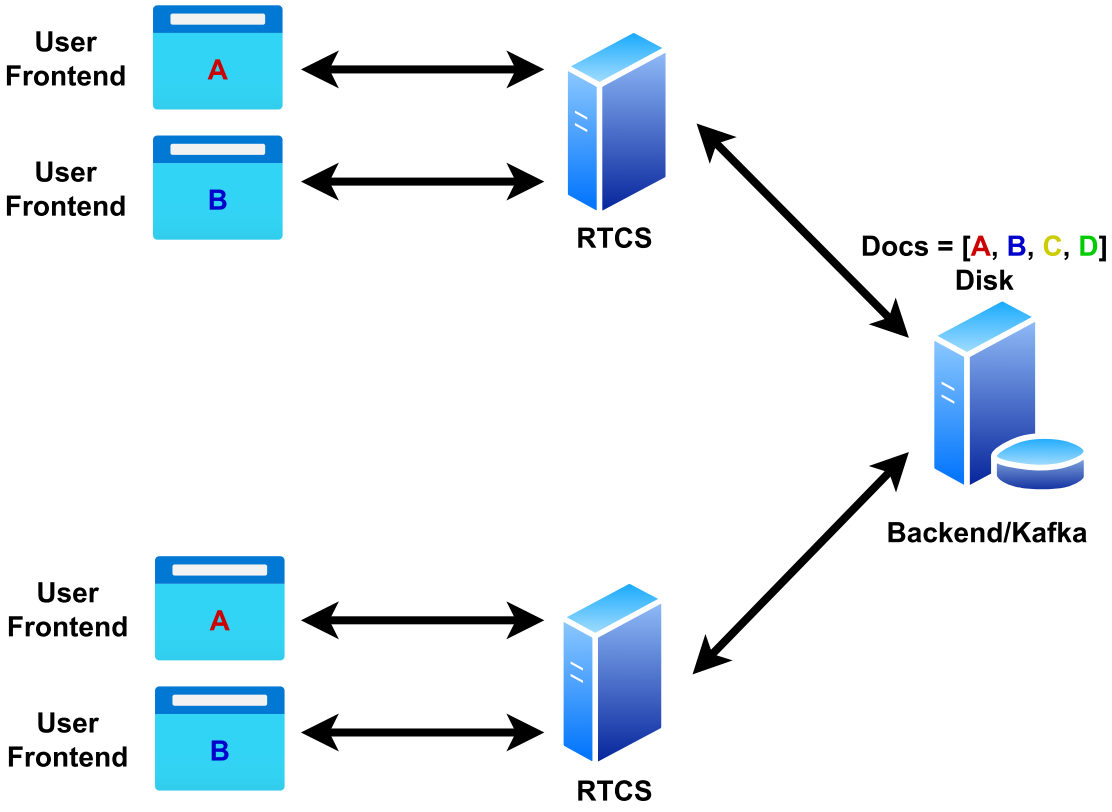


Figure 4.5: Scaling the RTCS as stateless instances with Apache Kafka

[5]https://kafka.apache.org/

Figure 4.5 shows the scaling of the real-time collaborative editing feature using stateless RTCS instances and a log-based message broker like Apache Kafka. Since all the state of the document is handled either in the frontend or in the log-based message broker at the backend, the RTCS only acts as a stateless proxy and can easily be horizontally scaled. Kafka serves as the persistence layer for the sequence of document updates in the backend that offers an API to produce and consume updates. Apache Kafka runs on a horizontally scalable cluster. It manages its own distribution and rebalancing of topics/partitions among nodes and uses replication to provide fault tolerance.

To summarize, this approach appears to be highly scalable while also offering elasticity. However, there are downsides for using this approach in the context of QDAcity. First of all, Apache Kafka uses a polling-based mechanism to consume messages. This may not be an issue for most use cases of Apache Kafka. However, as has been stated in subsection 2.2.2, a bidirectional communication protocol like WebSockets would be preferred to offer low latency ("real-time") reads of the most up-to-date state of a document. Secondly, while removing complexity in the form of statefulness from the RTCS instances, there would be a lot of new complexity introduced at the backend in the form of a distributed, log-based message broker like Apache Kafka as the persistence layer for documents. Since it was preferred to implement real-time collaborative editing for QDAcity without replacing major existing systems, a different approach was chosen that will be discussed in the next subsection.

4.5.4 Stateless Instances with Pub/Sub Service

As has been shown, both the approaches of scaling via stateful RTCS instances with a discovery/routing service and scaling via stateless RTCS instances with log-based message broker incur a significant complexity overhead to the horizontal scaling of the RTCS. However, to keep the horizontal scaling of the RTCS simple we wanted to enable horizontal scalability while keeping most of the behavior of the single RTCS instance from subsection 4.5.1 unchanged.

With the single instance approach of subsection 4.5.1, the RTCS loads the state of a document into its memory, to open a collaborative editing session. Users can perform real-time collaborative text editing, because they are connected via the collaborative editing session at the RTCS. We have seen in subsection 4.5.2 how additional instances can be added by partitioning the total state of the application (the collection of documents) and distributing them among the RTCS instances. However, we have also seen that the process of partitioning and distributing the total state of the application makes the RTCS instances de-facto stateful, leading to significant complexity overhead. Nevertheless, the complexity of partitioning, distribution, and request routing can be avoided by not just connecting users that are collaboratively editing via a shared RTCS instance, but

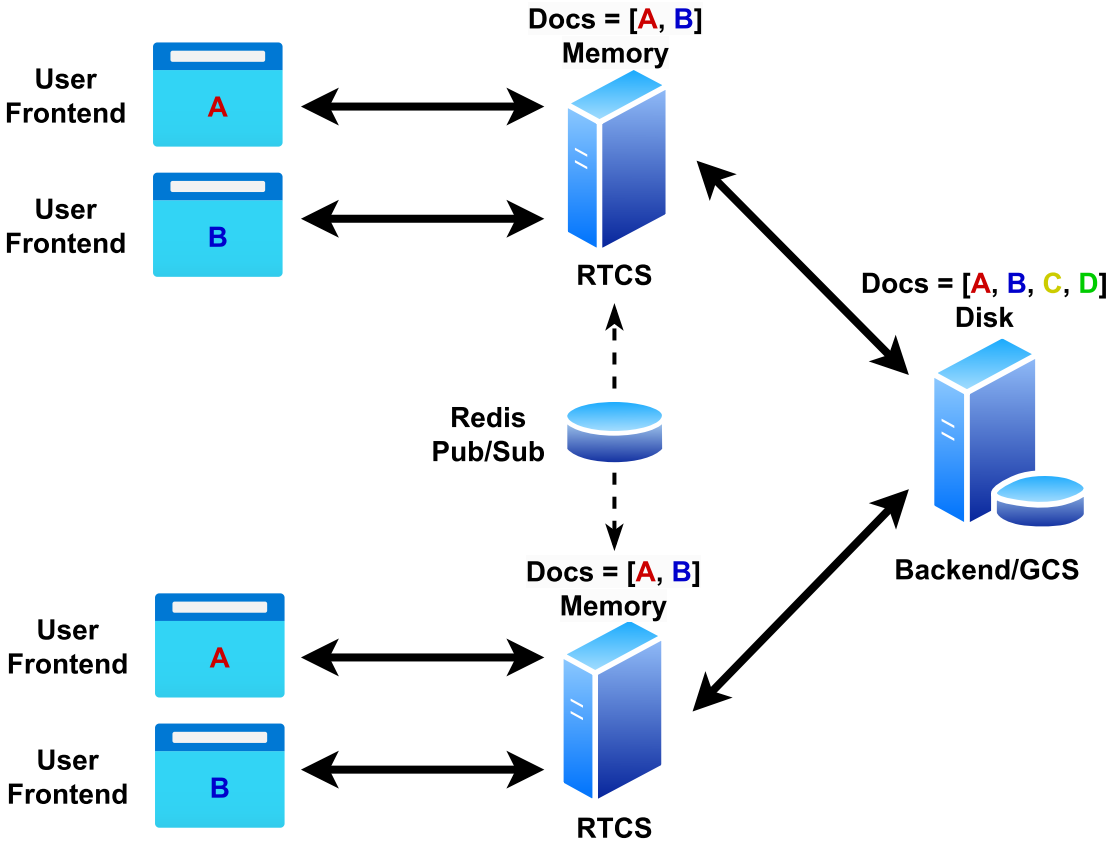


Figure 4.6: Reworked architecture of QDAcity

also interconnecting all RTCS instances. By doing so, all users can be connected to a certain collaborative editing session, regardless of which RTCS instance they are connected to. This approach can be seen in figure 4.6.

Figure 4.6 shows the interconnection of RTCS instances using Redis Pub/Sub[6]. Whenever an RTCS instance loads a document and starts a collaborative editing session it also creates a channel for this document in Redis Pub/Sub or subscribes to it if it already exists. All document edits that RTCS instances receive from users will also be published in the Redis Pub/Sub channel of this document so that users that are connected to different RTCS instances can participate in the same collaborative editing session. Unlike the producers and consumers from Apache Kafka, which generally use the classic request/response pattern of polling, Redis Pub/Sub uses stable TCP connections to provide fast delivery.

[6]https://redis.io/docs/manual/pubsub/

Using this approach, every RTCS instance can handle every user request, disregarding which documents the user wants to access. Consequently, the RTCS instances can be described as stateless, although each RTCS instance has to load the state of a document to memory to take part in its collaborative editing session. Furthermore, collaborative editing sessions stretching different regions around the globe could be implemented by interconnecting RTCS instances in different regions, close to the local users. With partitioning and stateful instances, all users of a collaborative editing session have to connect to the same RTCS instance. This leads to high latency for users in a distant geographical region. By interconnecting RTCS instances using Redis Pub/Sub users could always be connected to a RTCS instance in their own region. This would enable low latency at least among the local clients connected to the local instance.

To summarize, this approach enables horizontal scalability and elasticity with technically stateless RTCS instances and without major restructuring of the backend. It is the generally recommended approach for initial horizontal scaling in the community of the used CRDT implementation. Nonetheless, there are some downsides. First of all, the reduced complexity by avoiding stateful partitioning leads to inefficient redundancy in the main memory load of the RTCS instances, compared to the approach using stateful partitioning. As can be seen in figure 4.6, every RTCS instance that is participating in the collaborative editing session has to keep the document in the main memory. Additionally, the detour over Redis Pub/Sub will add some latency to the distribution of changes between users that are not connected to the same RTCS instance. Furthermore, a single Redis Pub/Sub instance will eventually turn out to be a bottleneck, once the load has reached a certain threshold. Unlike Apache Kafka, a Redis cluster does not inherently partition the total Pub/Sub load among Redis instances (Mor, 2018). Instead, partitioning of Pub/Sub for a Redis Cluster requires manual implementation. Nonetheless, the number of operations per second that a single Redis Pub/Sub instance is able to handle should be around 100.000 or higher[7].

Eventually, we decided to use this approach. This is mainly due to the combination of promising scalability and elasticity while being less complex to implement in the short term compared to the previously discussed approaches.

[7] https://redis.io/docs/management/optimization/benchmarks/

5 Design and Implementation

This chapter describes the details of the design and implementation of the new real-time collaborative editing feature of QDAcity. First, the most important libraries used for the implementation are introduced in section 5.1. Subsequently, implementation-specific processes and details of the different parts of the system are described in section 5.2.

5.1 Libraries

The main libraries used for the implementation of real-time collaborative editing for QDAcity are Yjs[1], Slate-Yjs[2] and Hocuspocus[3]. All three of these are open-source and MIT-licensed.

Yjs is a Javascript-based CRDT implementation that has grown popular in recent years. Its conflict resolution algorithm has first been described in a conference paper (Nicolaescu et al., 2016). Yjs is an operation-based CRDT implementation (also called Operation-based Commutative Replicated Data Type). This means that the global state converges by exchanging update messages containing only edits (Shapiro et al., 2011). This is contrary to state-based CRDT implementations (also called State-based Convergent Replicated Data Type), which converge by exchanging and merging the whole state of the CRDT. By only exchanging edits, operation-based CRDT implementations like Yjs possibly use significantly less network traffic compared to state-based CRDT implementations.

Although the advantages of CRDTs for synchronizing data in a distributed system are clear, they have been under scrutiny as a component for data-intensive real-world applications, due to their large memory overhead (Jahns, 2020). However, unlike automerge[4] (a different CRDT implementation that can be used for real-time collaborative text editing), Yjs does not attach meta data to every single

[1] https://github.com/yjs/yjs
[2] https://github.com/bitphinix/slate-yjs
[3] https://github.com/ueberdosis/hocuspocus
[4] https://automerge.org

character by default. Instead, Yjs implements some crucial optimizations for memory overhead and performance, e.g. merging sequential inputs to a single element[5]. The optimized performance of Yjs compared to automerge can be examined by executing a benchmark[6]. The success of this approach led to work-in-progress ports of Yjs from Javascript to C# and Rust.

Yjs provides so-called shared types like `Y.Map`, `Y.Array`, `Y.XmlElement`, and `Y.Text`. These can be added to a so-called `Y.Doc` that can be synchronized among clients. Within a `Y.Doc`, you can nest shared types and execute transactions in order to bundle changes of the contained shared types. In general, every data structure that can be encoded as JSON or as an `Uint8Array`[7] can be mapped to a shared type and thus synchronized between clients using Yjs. Additionally, ephemeral awareness information about the shared editing session can be distributed as part of a `Y.Doc`.

Yjs is a modular library. The synchronization of a `Y.Doc` among multiple clients is handled by a so-called network provider. For Yjs, multiple network provider implementations exist using different communication protocols like WebSockets[8], WebRTC[9], Dat[10], or libp2p[11]. In order to use Yjs for real-time collaborative text editing, multiple editor bindings exist that map the text body of a Javascript-based text editor to Yjs. One of these is Slate-Yjs which provides a text editor binding for Slate (the editor used for editable documents in the coding editor of QDAcity). It also provides a mechanism for integrating the awareness information of Yjs into the editor. Additionally, there is an IndexedDB[12] provider[13] for Yjs that can be used to persist local changes in the browser until synchronization for offline capabilities.

Hocuspocus is a Typescript-based WebSocket server implementation for Yjs built upon the WebSocket provider of Yjs. It provides a framework that handles WebSocket connections, collaborative sessions, configuration, etc. It also provides hooks for certain events that can be used with extensions. Table 5.1 shows a summary of the Hocuspocus hooks that are available. Via custom extensions, custom event handlers can be attached to these hooks to execute custom code at the right time. When the event handler is called, relevant information and objects are provided as parameters.

[5] https://text-crdt-compare.surge.sh
[6] https://github.com/dmonad/crdt-benchmarks
[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array
[8] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[9] https://webrtc.org
[10] https://datprotocol.com
[11] https://libp2p.io
[12] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
[13] https://docs.yjs.dev/getting-started/allowing-offline-editing

Hook	Description
beforeHandleMessage	Before handling a message
onConnect	When a connection is established
connected	After a connection has been establied
onAuthenticate	When authentication is required
onAwarenessUpdate	When awareness changed
onLoadDocument	When a new document is created
onChange	When a document has changed
onDisconnect	When a connection was closed
onListen	When the server is intialized
onDestroy	When the server will be destroyed
onConfigure	When the server has been configured
onRequest	When a HTTP request comes in
onStoreDocument	When a document has been changed
onUpgrade	When the WebSocket connection is upgraded

Table 5.1: Hocuspocus hooks that can be attached with custom event handlers by using a custom extension. (Source: https://tiptap.dev/hocuspocus/server/hooks)

Hocuspocus also comes with a set of extensions that use these hooks to add additional functionality or simply demonstrate the behavior of extensions for Hocuspocus. A summary of existing extensions is shown in table 5.2

Extension	Description
Database	A generic database driver that is easily adjustable to work with any database.
Monitor	A beautiful dashboard to monitor and debug your Hocuspocus instance.
Redis	Scale Hocuspocus horizontally with Redis.
Logger	Add logging to Hocuspocus.
Webhook	Send document changes via webhook to your API.
Throttle	Throttle connections by IPs.

Table 5.2: Provided Hocuspocus extensions.
(Source: https://tiptap.dev/hocuspocus/server/extensions)

Noteworthy, the Monitor extension provides a Vue frontend dashboard to monitor basic metrics of the Hocuspocus server. These include CPU usage, memory usage, connected clients, existing collaborative editing sessions, etc. The Redis extension can be used to connect the Hocuspocus server to a Redis Pub/Sub instance as discussed in subsection 4.5.4. The Throttle extension can be used to throttle client connections to prevent misuse. Additionally, Hocuspocus provides

a Vue[14] frontend called Playground that can be used for testing the functionality of the Hocuspocus server during development. When injecting extensions into a Hocuspocus instance, the extensions themselves can also be given configuration parameters to control their behavior.

This usage of extensions could be described as an application of the Inversion of Control-pattern (Fowler, 2005). This way, the Hocuspocus framework gives up control to a developer's custom code whenever decisions need to be made (like deciding whether a user is authorized to access a certain resource), or external dependencies need to be addressed (like a database for the persistence of documents). If multiple event handlers are attached to an event hook, the order of execution of event handlers can be controlled by assigning a priority value to the event handler. By using custom extensions to inject these event handlers into the Hocuspocus framework at initialization, the code of the Hocuspocus framework is strictly separated from the custom code injected into it. This enables the Hocuspocus framework to be maintained and updated independently from an application's custom code.

Yjs and the libraries building upon it were chosen for the CRDT-based real-time collaborative editing feature of QDAcity due to its great performance, active community, modular and open structure, versatile applicability, existing documentation, and the growing ecosystem of Yjs-based libraries like Slate-Yjs and Hocuspocus. Yjs is in use and sponsored by multiple commercial products as well as recommended by the global software consultancy company Thoughtworks[15] in 2022.

5.2 Implementation

In the RTCS, a Hocuspocus server instance is initialized and injected with configuration[16] parameters. An example is a debounce setting that controls the intervals of persisting documents to the backend. The most important extensions used are a custom implemented QDAcity-specific extension and the Redis extension. The Redis extension uses various event hooks to handle the connection to the Redis Pub/Sub instance, as well as producing and consuming document updates. Apart from Yjs, the Hocuspocus server and the custom QDAcity extension for Hocuspocus are the main parts of the implementation of the real-time collaborative editing feature for QDAcity.

[14] https://vuejs.org/

[15] https://www.thoughtworks.com/de-de/radar/languages-and-frameworks?blipid=202210047

[16] https://tiptap.dev/hocuspocus/server/configuration

The QDAcity extension uses the provided hooks (see table 5.1) to implement QDAcity-specific behavior. It contains code that handles authorization, initialization, loading, and storing of documents, as well as loading and persisting changes of the code system to the backend. Authorization of a user request is decided, by using a bearer token from the frontend that is received at the initialization of the WebSocket connection. When a document/code system is requested from the frontend, Hocuspocus checks whether there is an existing editing session for this document/code system. If not, it emits the onLoadDocument event to get the document/code system and initialize a new session.

In case of a document being requested, the onLoadDocument event handler in the QDAcity extension then tries the download the `Y.Doc` of the requested document from its Google Cloud Storage bucket. When the frontend requests a newly created document that has not been initialized yet, the QDAcity extension creates a new `Y.Doc` and fills it with an initial empty Slate paragraph and text node. This is a necessary requirement for newly created Slate documents. In order to execute this step exactly once, it happens at the RTCS instead of the Slate editor instance in the frontend. If a document is requested that only exists in the HTML representation that has been used for QDAcity so far, it converts it from its HTML representation to the new CRDT representation (`Y.Doc`). In case of a code system being requested, the RTCS acquires the necessary data by requesting it from the backend and then builds a `Y.Doc` for the code system by mapping the objects on shared types. After loading the requested resource, the Hocuspocus server provides the newly connected client with the requested `Y.Doc` by encoding the whole state of the `Y.Doc` as an update operation and delivering it to the newly connected client. As a result, the client is now synchronized with the collaborative editing session and can start working on the `Y.Doc`.

During the collaborative editing session, Hocuspocus calls the onStoreDocument event handler at regular intervals to regularly persist the document/code system to the backend. In the case of it being a text document, this means uploading the `Y.Doc` containing the text document in its (binary encoded) native CRDT representation to a Google Cloud Storage bucket. For a code system, this means checking whether the data in the `Y.Doc` has changed since the last call of onStoreDocument. If yes, the updated objects from the code system `Y.Doc` will be persisted to the backend using a batch request. When the last user leaves a collaborative editing session, the onStoreDocument event handler is called a last time before the session is cleaned up and garbage collected.

In the frontend, Hocuspocus provider objects (based on the WebSocket provider of Yjs) are used in the React component of the Slate text editor and the React context component that provides the code system information to the components using it. These objects connect and synchronize a locally initialized `Y.Doc` using the Hocuspocus server. In the case of the text editor component, Slate-Yjs is

injected as an extension for the Slate editor upon its initialization and connected to the local `Y.Doc`. The behavior of the Hocuspocus provider can be configured using various parameters. A URL parameter is used by the provider to setup the connection to the server. Various parameters exist to configure a retry policy for attempting to reconnect to the server in case of a disconnect. Using a token parameter, the user token is passed to the provider for authentication and authorization of the user at the Hocuspocus server. A name parameter is used to identify collaborative editing sessions by ID. Additionally, the Hocuspocus provider also emits events and supports the attachment of event handlers. Table 5.3 shows a list of events emitted by a Hocuspocus provider. These events are used to reflect changes in the connection life cycle and the connected `Y.Doc` in the frontend. For example, the synced event is used to trigger displaying the received data in the frontend after synchronization with the server. Afterwards, the message event is used to update the state of the view on the data by rerendering a component once relevant data has been edited by a different client.

Event	**Description**
open	When the WebSocket connection is created.
connect	When the provider has successfully connected to the server.
authenticated	When the client has successfully authenticated.
authenticationFailed	When the client authentication was not successful.
status	When the connections status changes.
message	When a message is incoming.
outgoingMessage	When a message will be sent.
synced	When the Y.js document is successfully synced (initially!).
close	When the WebSocket connection is closed.
disconnect	When the provider disconnects.
destroy	When the provider will be destroyed.
awarenessUpdate	When the awareness updates
awarenessChange	When the awareness changes
stateless	When the stateless message was received.

Table 5.3: Events emitted by the Hocuspocus provider objects in the frontend. (Source: https://tiptap.dev/hocuspocus/provider/events)

At the backend, a new DocumentType and a new class CollaborativeTextDocument (inheriting from TextDocument) were added. It contains and manages the additional meta data that is necessary for loading and storing the `Y.Doc` from and to Google Cloud Storage (GCS). Furthermore, it implements some methods for backend workflows handling the data of the new subclass, e.g. for cloning of documents/projects.

6 Evaluation

This chapter revisits the requirements listed in chapter 3 and individually evaluates whether the requirements have been fulfilled. The requirements are assessed as fulfilled, partially fulfilled, or not fulfilled. Analogous to chapter 3, we will start with the functional requirements in section 6.1. Afterwards, we will evaluate the fulfillment of the nonfunctional requirements in section 6.2.

6.1 Functional Requirements

FR-1: The RTCS shall be extended by a new CES that generally enables real-time collaborative editing for users of QDAcity.

> **FR-1.1:** In order to keep overhead for maintenance and deployment low, the CES shall be an extension of the QDAcity RTCS.
>
> **FR-1.2:** The existing Slate[1] text editor in the QDAcity frontend shall be extended to connect to the CES and make use of the real-time collaborative editing functionality that the CES provides.
>
> **FR-1.3:** In order to provide a consistent user experience in the coding editor, the CES or reworked RTCS shall be able to keep the code system synchronized with the text document.
>
> **FR-1.4:** The CES shall only process the operations of authenticated and authorized users.

The implemented CES enables real-time collaborative editing using Yjs. It is an extension of the RTCS and connects to the Slate text editor in the frontend via the Slate-Yjs plugin. In order to keep the code system synchronized with the text document a collaboration solution has been implemented with Yjs that can synchronize both the code system and the text document. Changes to both of these will be distributed and pushed to other users' frontends in real time. A

[1] https://docs.slatejs.org/

bearer token from the frontend is being used to authenticate and authorize users at the initial connection to the CES.

The requirement FR-1 has been fulfilled.

FR-2: The real-time collaborative editing feature that the CES provides shall enable efficient document synchronization across multiple users.

> **FR-2.1:** As soon as a user requests a document that is not currently being edited, the CES shall load this document to memory and provide it to requesting users.
>
> **FR-2.2:** As soon as a user edits a document, the CES shall accept and distribute these edits to all other concurrent editors of this document in real-time, by using a bidirectional communication protocol.
>
> **FR-2.3:** As long as one or multiple users are concurrently editing a document, the CES shall prevent write conflicts by continuously converging the state of the document.
>
> **FR-2.4:** As long as a document is actively being edited, the CES should regularly persist the current state of the document to the backend.
>
> **FR-2.5:** As soon as the last editing user of a document has disconnected, the CES should persist the most up-to-date state of the document to the backend.
>
> **FR-2.6:** As long as a document is not actively being edited, the CES should not hold it in memory.

The implemented CES only keeps text documents and code systems in memory that are actively being worked on. WebSocket connections are used to distribute incoming edits among the connected clients. Using Yjs as a CRDT-based solution, the state of the document is being merged continuously. This applies no matter the order of updates or the delay of distributing updates among the users. As long as the `Y.Doc` of a document/code system is not discarded, incoming updates can be applied and the `Y.Doc` will globally converge, thus preventing write conflicts. Using a debounce configuration, the state of a document is persisted at regular intervals, preventing large data loss in case of a fault at the RTCS. When a collaborative editing session is closed, the CES removes the `Y.Doc` from memory, freeing up resources for other collaborative editing sessions.

The requirement FR-2 has been fulfilled.

6.2 Nonfunctional Requirements

NFR-1: The CES should be designed in a way that can be deployed and run in the environment that Google Cloud Run provides.

> **NFR-1.1:** Google Cloud Run offers a maximum of 32 gigabytes of main memory per instance.[2]
>
> **NFR-1.2:** Google Cloud Run offers a maximum of 1000 concurrent WebSocket connections per instance.[3]
>
> **NFR-1.3:** Google Cloud Run offers a maximum of 60 minutes per WebSocket connection before timing out.[4]

The CES, as part of the RTCS, can be deployed and run in the environment that Google Cloud Run provides. The constraints of the hardware capacity provided by Google Cloud Run has been mitigated by choosing an optimized CRDT implementation in the form of Yjs. The constraint of a capped connection count per Google Cloud Run instance has been addressed by the capability of the CES to scale horizontally. The automated timeout of the WebSocket connection after 60 minutes has been addressed by using a connection provider in the frontend that automatically attempts to reconnect after an unexpected disconnect. To summarize, the requirement was met with the proviso that documents of human-authored size are used.

The requirement NFR-1 has been fulfilled.

NFR-2: The CES shall be designed in a way that provides compatibility for existing features and possible feature extensions in the future.

> **NFR-2.1:** The CES shall be implemented in a way that does not break the existing functionality of QDAcity.
>
> **NFR-2.2:** The CES shall be designed in a way that could support syncing additional document formats in the future.
>
> **NFR-2.3:** The CES should be designed in a way that is compatible with offline editing.

The CES was generally implemented in a way that does not break the existing functionality of QDAcity. The coding feature has been reworked to enable it to be compatible with the new system. By also persisting documents in the initial HTML format, derived from the `Y.Doc`, the backend can keep performing its document analyses without extensive adaptations required. However, the document

[2] cloud.google.com/run/quotas
[3] cloud.google.com/run/docs/about-concurrency
[4] cloud.google.com/run/docs/triggering/websockets

analysis code has not yet been adapted to the new mechanism of embedding coding information in a text document. The new document class CollaborativeTextDocument has been implemented in a way that converts existing text documents at their first request through the CES. The capability of Yjs to generally synchronize and converge JSON-based data enables the option of extending the real-time collaborative editing feature with additional document formats in the future. With the proviso that features that have been implemented in parallel to the CES are not considered here, the requirement is seen as fulfilled.

The requirement NFR-2 has been fulfilled.

NFR-3: The CES should be designed in a way that provides a degree of scalability and elasticity, while not deviating from the GCP-based services that are already in use by QDAcity.

> **NFR-3.1:** The CES shall be designed in a way that is horizontally scalable to cope with increasing load.
>
> **NFR-3.2:** The CES should be designed in a way that can make use of the elasticity that its underlying GCP services provide.

By implementing the architecture presented in subsection 4.5.4, the RTCS can be horizontally scaled using stateless instances and Redis Pub/Sub. Google Cloud Run remains the hosting service for the modified RTCS including the newly implemented CES. Due to the stateless nature of the RTCS instances, the newly implemented real-time collaborative editing feature is able to benefit from its automated scaling and elasticity capabilities. Thus, we assess this requirement as fulfilled.

The requirement NFR-3 has been fulfilled.

NFR-4: The CES shall be designed in a way that provides sufficient performance.

> **NFR-4.1:** The CES should be designed in a way that can support the real-time collaborative editing of a single document by at minimum five users.
>
> **NFR-4.2:** The CES should be designed in a way that can support the real-time collaborative editing of at minimum 50 average-size documents on a single CES instance.
>
> **NFR-4.3:** In order to provide a satisfying degree of responsiveness, the average latency of distributing edits between connected users should be less than 1000 milliseconds.

With manual testing, the minimum user count was confirmed. At this point, the number of documents that a single CES instance can handle could only be estimated. With local tests, a real-world human-authored document with an extent of 15-20 pages is assessed as requiring a single-digit number of megabytes of main memory. Thus, 50 documents of this size should allocate a three-digit number of megabytes of main memory. When a single RTCS instance reaches its cap of 1000 concurrent WebSocket connections, assuming every connected user edits a different document with an extent of 15-20 pages, the RTCS is expected to allocate a single-digit number of gigabytes of main memory. This should be well within the required performance. The latency is mainly affected by the distances between the data center and the user, but judging by the benchmark results[5] of Yjs, it should not be an obstacle in achieving a satisfying degree of responsiveness. All in all, we expect this requirement to be fulfilled, however, more practical evaluations are needed to confirm the assessments that are based on local executions and benchmarks. Thus, we evaluate this requirement as partially fulfilled.

The requirement NFR-4 has been partially fulfilled.

NFR-5: The implementation of the CES should be designed in a way that enables sufficient maintainability.

> **NFR-5.1:** The implementation of the CES should be designed in a way that is well documented internally (commented code) as well as externally (in the QDAcity Wiki).
>
> **NFR-5.2:** The implementation of the CES should be designed in a way that can easily be extended.
>
> **NFR-5.3:** The implementation of the CES should be designed in a way that supports the creation of acceptance tests.

The implementation of the CES is documented internally. The used libraries Yjs, Slate-Yjs, and Hocuspocus also provide comprehensive documentation. While no dedicated pages have been written for the QDAcity Wiki about the implementation of the CES, parts of this thesis will be used and modified for the QDAcity Wiki. The event hook-based framework of Hocuspocus provides a simple way of implementing and injecting extensions, significantly facilitating the extendability of the CES. In general, it should be possible to create acceptance tests for the real-time collaborative editing feature using established tools like Selenium. However, this has not been tackled in the context of this thesis. To summarize, this requirement has been partially fulfilled.

The requirement NFR-5 has been partially fulfilled.

[5] https://github.com/dmonad/crdt-benchmarks

NFR-6: The CES should be designed in a way that is reliable and fault-resistant.

> **NFR-6.1:** The CES should be designed and deployed in a way that can consistently and autonomously recover from crashes and disconnects.

In case of failure of an RTCS instance, its stateless nature enables the client to simply reconnect to a different instance. It is not necessary for the client to wait until some partitioning or replication management system has made the affected partition available on a different RTCS instance. Upon reconnection, the local changes of the client that have not reached the RTCS yet will be synchronized to the new session and the local changes of the user will be persisted. However, the RTCS does not immediately persist every keystroke to the backend and does not write changes to local disk. Instead, it uses a debounce setting to persist changes to the backend at regular intervals. Thus, the possibility of data loss exists. If no other user is in the collaborative editing session to receive the changes, and the RTCS crashes with non-persisted changes, then the new changes only exist on the client of the user. If the user quits without reconnecting and sharing his local changes with a different RTCS instance, the local changes will be lost. This is where an offline collaborative editing feature for the frontend client could help by locally persisting these changes until the user reconnects to an RTCS instance at some point in the future. However, it cannot be guaranteed that any user ever reconnects. Nonetheless, given a debounce setting of e.g. half a minute, the expected data loss should be negligible. All in all, it is definitely an improvement on the initial system. Since the fault-tolerance of the CES has only been theoretically evaluated and not practically tested, we evaluate this requirement as partially fulfilled.

The requirement NFR-6 has been partially fulfilled.

NFR-7: The CES should be designed in a way that benefits usability.

> **NFR-7.1:** The CES should be designed in a way that supports intuitive understanding by its users by displaying feedback for connection problems.
>
> **NFR-7.2:** The CES should be designed in a way that supports intuitive understanding by its users by displaying text editing awareness information.

Awareness information has been implemented as part of Yjs and Slate-Yjs. Connection feedback exists for the initial RTCS, but does not include the connection to the CES part of the RTCS. This could be unified by attaching the connection information to the synchronization of the code system. However, connection feedback has not been implemented yet (apart from console output). Additionally, concrete practical tests including user feedback are required to test the usability.

The requirement NFR-7 has been partially fulfilled.

7 Discussion

In this chapter, further noteworthy aspects of the development process will be discussed. Section 7.1 deals with the consideration of using a P2P-based network model instead of the traditional client/server pattern. Subsequently, section 7.2 describes the consideration of implementing a hybrid between the initial and the reworked model for collaborative editing sessions. Lastly, 7.3 gives a general impression of the difficulties and aids of the development process and proposes some future work.

7.1 Network Model

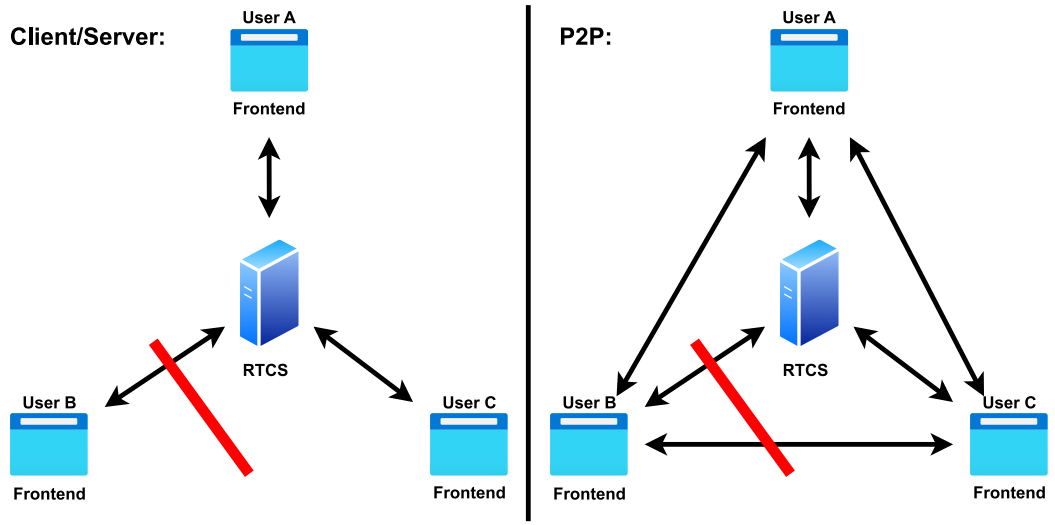


Figure 7.1: Comparison of synchronization of a document via client/server or P2P network mode. Equivalent network partitions in red.

CRDTs have the capability to globally converge in a P2P network, without requiring a central server. Thus, the question arises whether the new iteration of QDAcity should rely on a client/server pattern or P2P network to synchronize the state of a document between simultaneously editing users. A P2P approach may

have some advantages over the traditional client/server pattern. A comparison between both approaches is displayed in figure 7.1.

The client/server pattern is on the left side and the P2P approach is on the right side of figure 7.1 Both network models experience the same network partition, displayed as red bars. On the left side, user B is completely disconnected from the RTCS, due to the network partition. On the right side, user B could use the P2P connection to user A to still be connected to the other participants, exchange document edits, and remain in sync with the rest of system. Thus, the P2P approach could potentially provide better availability of QDAcity when facing network partitions compared to the client/server pattern.

However, P2P-topologies are usually a good fit for networks where all participants are equal. In the context of this example, the participants are not equal. Only edits that arrive at the RTCS will be persisted to the backend. Additionally, as the vendor of QDAcity, only the RTCS can be relied on to stay online (excluding incidents of faults or network partitions). Participating users could go offline and never return at any point in time. Consequently, if user B relies on user A forwarding the edits of user B to the RTCS, as seen on the right side of figure 7.1, user A could go offline at any time. Thus, terminating user B's connection to the rest of the system. Thus, for user B, relying on other users to transmit its edits to the RTCS could result in an unstable and intransparent experience.

Providing connection indicators to the user appears to be an easy fix for the transparency issue. In the case of a client/server pattern, this would be a binary indicator. It would indicate whether a connection exists between a user and the RTCS or not. For the P2P approach, a third option would be needed. This option would indicate that an indirect connection via other users exists but could drop at any time, as soon as a required user goes offline.

To add to this example, two users, i.e. users A and B, may both experience a loss of connection from the RTCS while editing the same document. This could possibly be due to some fault on the side of the RTCS. In this case, users A and B could still exchange updates and continue real-time collaborative editing over their local P2P connection. However, if both users go offline indefinitely after making their changes, their version of the document would never be persisted. The exchange of edits between users A and B could delude them into believing they are still connected to the RTCS when they actually are not. Having a connection indicator indicating that no connection to the RTCS exists, yet there are still edits from another user arriving, could again lead to an intransparent and confusing experience for users.

In the end, we decided the potential availability advantage of the P2P approach does not justify a more complex system, which could lead to a possibly confusing or unstable user experience. Thus, we decided to keep the traditional client/server pattern and a binary connection indicator.

7.2 Hybrid Model for Collaborative Editing Sessions

In its new state, text documents are being edited using CRDTs and the RTCS, disregarding whether there are actually multiple users collaboratively editing a document or there is only one user making changes. During the development process, it was evaluated whether to keep the old system of making and persisting changes of a document (as explained in section 4.2) in place as long as only a single user is editing a document. A collaborative editing session at the RTCS would only be created once at least two users start simultaneously editing the same document. Another possibility would be to allow users to choose whether they want to create/join a collaborative editing session or work on their own diverging local version (without synchronization) instead. Thus, basically opting out of the real-time collaborative editing system.

Since in a real-time collaborative editing session every keystroke is communicated to the RTCS, the main advantage of a hybrid approach would be the possibility of reducing traffic and memory load of the RTCS as long as the real-time collaborative editing feature is not required. However, we decided not to go with a hybrid approach of the old and new system due to the following reasons:

- Running two approaches concurrently leads to increased effort in implementation and maintenance. Additionally, allowing users to permanently diverge their local state of a document from the synchronized version (basically branching) increases the complexity of reasoning about the state of the documents in a project.
- Whenever a second user joins a single user in editing a document, the editing session needs to be migrated from the old approach to the new approach of collaborative editing (or vice versa), possibly leading to disruptions in the editing process.
- Since with the new approach, every keystroke is sent to the RTCS and expected to be persistent, switching between approaches might confuse a user's expectation in terms of when his changes are persisted. A user that usually edits in real-time collaborative sessions might expect his changes to also be immediately persisted when he is editing a document alone. This could lead to unexpected data loss if the user loses connection before his

changed version of the document is persisted to the backend, leading to a bad user experience. The immediate communication and persistence of every keystroke can be seen as a feature for data loss protection disregarding the number of concurrent users editing the same document.

7.3 Development Process and Future Work

The agile development process mainly consisted of a tight feedback loop between conducting research, prototyping, new insights, and feedback from the supervisors. In general, the development process was shaped significantly by the number of open questions at the beginning of the thesis process. This required a comprehensive research process of examining existing solutions, evaluating the most suitable approaches of real-time collaborative editing for QDAcity, and how to additionally synchronize the code system. This similarly applies to the design of the horizontal scaling capability.

Moreover, the fact that the newly implemented feature affects the frontend, RTCS, backend, and Google Cloud Storage leads to a significant threshold of getting acquainted with the existing system, before being able to draw conclusions from the comprehensive research process. The effort of getting acquainted with the existing system can be traced back to the number of different technologies, frameworks, and cloud services used in the different domains of the system. Due to the number of features implemented by various students during their thesis, many files of the frontend that were relevant for this thesis have reached significant lengths.

Furthermore, most of the new technologies that have been used for the implementation of real-time collaborative editing for QDAcity are based on Typescript and functional React components. Thus, the still widespread use of plain Javascript-based class components in the frontend and Javascript in the RTCS complicated the adoption of these technologies. The architectural decisions have been influenced by the approach of taking incremental steps and the preference of sticking with the systems that are already in use by QDAcity. The collaboration with the supervisors, the extensive test suite, and the comprehensive documentation in the QDAcity repository provided tremendous support for the development of this feature.

Future work that could build upon the results of this thesis includes the following:

- Currently, the backend code for analyzing projects and documents has not yet been adapted to the new mechanism of embedding coding information in a collaborative text document. This is required for full compatibility of existing features upon rollout of the real-time collaborative editing feature. Also, the existing deserialization function for documents (from HTML to

Slate nodes) needs to be adapted to also convert the coding information embedded in the document to its new representation.

- The presentation of awareness information is currently split between the old and new approach of the RTCS for distributing information. These could and should be unified using Yjs to condense the use of multiple WebSocket connections to multiplexing[1] on a single WebSocket connection using Hocuspocus. The multiplexing feature for Hocuspocus is currently (as of April 2023) in development. It should be used for QDAcity since the number of WebSocket connections per Google Cloud Run instance is limited to 1000. The awareness feature of Yjs can be used to distribute ephemeral information that will not be persisted as part of the `Y.Doc`. Via private collaborative sessions, it is possible to only push information to specific users.
- Yjs supports offline collaborative editing functionality but it has not been implemented in the context of this thesis, since aside from the synchronization of the text document, the focus was on the synchronization of the code system. The offline collaborative editing feature could and should be implemented as part of future iterations of QDAcity. Using the Y-indexeddb[2] provider to store local edits persistently in the browser during a disconnect, this feature should be possible to implement with moderate effort. Note that for offline collaborative editing, the involved `Y.Doc`-objects need to be persisted to allow retrospective convergence.
- Apart from the code system, other document types and existing features of QDAcity that are currently not synced could also benefit from real-time synchronization. Examples include document types like drawing boards or calculation sheets.
- As the current approach of horizontal scaling is bottlenecked by the single Redis Pub/Sub instance, one could manually implement partitioning the Redis Pub/Sub channels for a Redis cluster or implement one of the other approaches to horizontal scaling that have been described in section 4.5.
- Parts of the nonfunctional requirements have been implemented but not tested yet, e.g. usability. To comprehensively evaluate the fulfillment of these requirements, more practical tests including user feedback are required.

[1] https://github.com/ueberdosis/hocuspocus/pull/484
[2] https://docs.yjs.dev/getting-started/allowing-offline-editing

8 Conclusion

This last chapter concludes and sums up the thesis. Chapter 1 motivated this thesis, set its objective, and described the structure of this thesis. We claimed that QDAcity as a platform for collaboratively conducting Qualitative Data Analysis (QDA) could benefit significantly from the implementation of a real-time collaborative editing feature. We set the objective of implementing a horizontally scalable, real-time collaborative editing feature for QDAcity.

Chapter 2 discussed and summarized the most relevant related work and research for this thesis. First, we briefly introduced QDAcity's coding editor, which is the component that shall be extended by the implementation of the new feature. Subsequently, we discussed the difficulties of real-time collaborative text editing and the approaches to handling these difficulties. We came to the conclusion that to enable real-time editing, a shift from strong consistency to strong eventual consistency is necessary. The approaches covered include Differential Synchronization (DS), Operational Transformation (OT), and Conflict-free Replicated Data Types (CRDTs). We concluded that OT and CRDTs are both capable solutions that are able to guarantee convergence. However, CRDTs offer more useful abstractions for convergence in a distributed environment. Lastly, we addressed the topic of scaling cloud-based web applications. We confirmed our assumption that a purely vertical approach to scaling is inherently limited and a horizontal scaling approach is preferable long-term. To implement horizontal scaling with reasonable effort, keeping application instances stateless is a significant aspect.

Chapter 3 formulated the collected requirements for the implementation part of this thesis. These were categorized into functional and non-functional requirements. The most significant requirements included the implementation of the new Collaborative Editing Service (CES) as part of the existing Real-Time Collaboration Service (RTCS). The feature should be able to synchronize various document formats and features of QDAcity. At the same time, it should be horizontally scalable while being based on the Google Cloud Platform (GCP) services that are already in use by QDAcity. Furthermore, the requirements emphasized compatibility, maintainability, performance, usability, and reliability.

Chapter 4 discussed the architectural adaptations that were made in the context of this thesis to achieve the desired functionality. Originally, the transmission of the document data happened directly between the frontend client and the backend via HTTP. Unlike before, the newly implemented architecture handles the transmission of documents via a WebSocket connection to the RTCS. Various design drafts were described. This includes the approach that was finally implemented using Redis Pub/Sub to interconnect stateless RTCS instances.

Chapter 5 describes the most significant used libraries (Yjs, Slate-Yjs, Hocuspocus) and QDAcity-specific implementation details of the new real-time collaborative editing feature. Yjs is a modular Javascript-based CRDT implementation that features some optimizations and can synchronize JSON data in general. Slate-Yjs is a library that maps the JSON-based content of a Slate text document on Yjs' CRDTs. Hocuspocus is a Typescript-based, extendable server implementation that uses WebSocket connections to synchronize collaborative editing sessions via Yjs. It offers hooks to inject custom behavior via custom event handlers and extensions. Furthermore, it was described how these libraries were used in the various domains of the QDAcity software architecture to implement the real-time collaborative editing feature.

Chapter 6 revisits and evaluates the fulfillment of the requirements as stated in chapter 3. We came to the conclusion that the most significant requirements of the functionality of the new feature were fulfilled. However, due to the research-heavy and technical topic of this thesis, the fulfillment of some requirements could only be confirmed based on theoretical estimates and analyses. Further practical testing and deployments are required to fully confirm these requirements.

Chapter 7 discussed considerations of further adaptations to QDAcity that were in consideration. These considerations dealt with a P2P-based network topology between clients and RTCS, as well as a hybrid model of the old and the new approach to editing documents. Since their advantages were evaluated to not justify the additional complexity, we ultimately decided against both of these mechanisms. Lastly, this chapter presented general considerations about the development process and proposed some future work. The suggestions for future work cover finishing the adaptation of the backend analysis tools to the new collaboration system, the unification of transmission of awareness information, offline collaborative editing functionality, additional collaborative features or document types beyond text documents, and practical tests for subjective requirements.

Concluding this thesis, we are satisfied with the implemented solution for QDAcity's real-time collaborative editing feature. There is still further work to be done and practical insights to be gained that build upon the implemented solution. However, the implemented solution combines the fulfillment of the most significant technical requirements with providing a synchronization platform that supports comprehensive future extensions of QDAcity's collaboration features.

References

Akhtar, W. (2022, March 15). *The Mother of All Demos*. Medium. Retrieved March 19, 2023, from https://medium.com/@wicar/the-mother-of-all-demos-719b3c666046

Alexei Baboulevitch. (2018, March 24). *Data Laced with History: Causal Trees & Operational CRDTs*. Archagon Was Here. Retrieved March 4, 2023, from http://archagon.net/blog/2018/03/24/data-laced-with-history/

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A view of cloud computing. *Communications of the ACM*, *53*(4), 50–58. https://doi.org/10.1145/1721654.1721672

Beck, K. (2000). *Extreme programming eXplained: Embrace change*. Addison-Wesley.

Boelmann, C., Schwittmann, L., Waltereit, M., Wander, M., & Weis, T. (2016). Application-Level Determinism in Distributed Systems, 989–998. https://doi.org/10.1109/ICPADS.2016.0132

Bourgon, P. (2014, May 9). *Roshi: A CRDT system for timestamped events*. Retrieved March 12, 2023, from https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events

Bryman, A., & Bell, E. (2011). *Business research methods* (3rd ed). Oxford University Press.

Creswell, J. W., & Creswell, J. D. (2018). *Research design: Qualitative, quantitative, and mixed methods approaches* (Fifth edition). SAGE.

Day-Richter, J. (2010, September 23). *What's different about the new Google Docs: Making collaboration fast*. Google Drive Blog. Retrieved March 9, 2023, from https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html

de la Vega, A., & Kolovos, D. (2022). An efficient line-based approach for resolving merge conflicts in XMI-based models. *Software and Systems Modeling*, *21*(6), 2461–2487. https://doi.org/10.1007/s10270-022-00976-4

Dey, I. (2005). *Qualitative data analysis: A user-friendly guide for social scientists*. Taylor & Francis e-Library
OCLC: 646796703.

Dutta, S., Gera, S., Verma, A., & Viswanathan, B. (2012). SmartScale: Automatic Application Scaling in Enterprise Clouds. *2012 IEEE Fifth International Conference on Cloud Computing*, 221–228. https://doi.org/10.1109/CLOUD.2012.12

Ellis, C. A., & Gibbs, S. J. (1989). Concurrency control in groupware systems. *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 399–407. https://doi.org/10.1145/67544.66963

Etherpad: A real-time collaborative editor for the web. (2011, March 26). Retrieved March 10, 2023, from https://github.com/ether/etherpad-lite/blob/48381de576909bc1db5c9dd30ae9495ec0ae55c1/doc/easysync/easysync-full-description.pdf

Fowler, M. (2005, June 26). *InversionOfControl*. martinfowler.com. Retrieved April 7, 2023, from https://martinfowler.com/bliki/InversionOfControl.html

Fraser, N. (2009a, January). *Writing: Differential Synchronization*. Retrieved March 9, 2023, from https://neil.fraser.name/writing/sync/

Fraser, N. (2009b). Differential synchronization. *Proceedings of the 9th ACM Symposium on Document Engineering*, 13–20. https://doi.org/10.1145/1600193.1600198

Gentle, J. (2016, August 18). *On: A Conflict-Free Replicated JSON Datatype*. https://news.ycombinator.com/item?id=12311984

Gilbert, S., & Lynch, N. (2012). Perspectives on the CAP Theorem. *Computer*, *45*(2), 30–36. https://doi.org/10.1109/MC.2011.389

Graue, C. (2015). Qualitative Data Analysis. *International Journal of Sales, Retailing and Marketing*, *4*(9), 5–14.

Hedkvist, P. (2021, February 28). *Introduction to Conflict Free Replicated Datatype*. The Startup. Retrieved March 12, 2023, from https://medium.com/swlh/introduction-to-conflict-free-replicated-data-type-959a944098c4

Herron, A. (2020, January 13). *Building real-time collaboration applications: OT vs CRDT*. Blueprint - Blog by Tiny. Retrieved March 3, 2023, from https://www.tiny.cloud/blog/real-time-collaboration-ot-vs-crdt/

Jahns, K. (2020, August 10). *Are CRDTs suitable for shared editing?* Kevin's Blog. Retrieved April 7, 2023, from https://blog.kevinjahns.de/are-crdts-suitable-for-shared-editing/

Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems* (First edition). O'Reilly Media
OCLC: ocn893895983.

Kleppmann, M. (2018, March 5). *CRDTs and the Quest for Distributed Consistency*. London. https://martin.kleppmann.com/2018/03/05/qcon-london.html

Kleppmann, M., & Beresford, A. R. (2017). A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, *28*(10), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: A Distributed Messaging System for Log Processing. Retrieved April 5, 2023, from https://www.semanticscholar.org/paper/Kafka-%3A-a-Distributed-Messaging-System-for-Log-Kreps/ea97f112c165e4da1062c30812a41afca4dab628

Kwan, A., Wong, J., Jacobsen, H.-A., & Muthusamy, V. (2019). HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres. *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 80–90. https://doi.org/10.1109/ICDCS.2019.00017

Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *21*(7).

Li, D., & Li, R. (2006). A performance study of group editing algorithms. *12th International Conference on Parallel and Distributed Systems - (ICPADS'06)*, *1*, 8 pp.-. https://doi.org/10.1109/ICPADS.2006.18

Lindholm, T. (2004). A three-way merge for XML documents. *Proceedings of the 2004 ACM Symposium on Document Engineering*, 1–10. https://doi.org/10.1145/1030397.1030399

Lorido-Botran, T., Miguel-Alonso, J., & Lozano, J. A. (2014). A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, *12*(4), 559–592. https://doi.org/10.1007/s10723-014-9314-7

Mihas, P. (2019, May 23). Qualitative Data Analysis. In *Oxford Research Encyclopedia of Education*. Oxford University Press. https://doi.org/10.1093/acrefore/9780190264093.013.1195

Mor, S. (2018). *Scaling Redis PubSub*. Retrieved April 5, 2023, from https://www.slideshare.net/RedisLabs/redis-day-tlv-2018-scaling-redis-pubsub

Nichols, D. A., Curtis, P., Dixon, M., & Lamping, J. (1995). High-latency, low-bandwidth windowing in the Jupiter collaboration system. *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, 111–120. https://doi.org/10.1145/215585.215706

Nicolaescu, P., Jahns, K., Derntl, M., & Klamma, R. (2016). Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types, 39–49. https://doi.org/10.1145/2957276.2957310

Oster, G., Molli, P., Urso, P., & Imine, A. (2006). Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, *0*. https://doi.org/10.1109/COLCOM.2006.361867

Peter Bourgon on CRDTs, Go at SoundCloud. (2015, January 15). Retrieved March 12, 2023, from https://www.infoq.com/interviews/bourgon-crdt-go/

Ptaszek, M. (2014, September 20). *Scaling League of Legends Chat to 70 million Players*. Retrieved March 12, 2023, from https://thestrangeloop.com/2014/scaling-league-of-legends-chat-to-70-million-players.html https://youtu.be/_jsMpmWaq7I

Pujol, J. M., Erramilli, V., Siganos, G., Yang, X., Laoutaris, N., Chhabra, P., & Rodriguez, P. (2010). The little engine(s) that could: Scaling online social networks. *Proceedings of the ACM SIGCOMM 2010 Conference*, 375–386. https://doi.org/10.1145/1851182.1851227

Rossi, F., Nardelli, M., & Cardellini, V. (2019). Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning. *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 329–338. https://doi.org/10.1109/CLOUD.2019.00061

Rupp, C., & SOPHIST-Gesellschaft für Innovatives Software-Engineering (Eds.). (2014). *Requirements-Engineering und -Management: aus der Praxis von klassisch bis agil* (6., aktualisierte und erw. Aufl). Hanser.

Seers, K. (2012). Qualitative Data Analysis. *Evidence Based Nursing*, *15*(1), 2–2. https://doi.org/10.1136/ebnurs.2011.100352

Shapiro, M., Preguiça, N., Baquero, C., & Zawirski, M. (2011). Conflict-free Replicated Data Types.

Sheikhan, M., & Ahmadluei, S. (2013). An intelligent hybrid optimistic/pessimistic concurrency control algorithm for centralized database systems using modified GSA-optimized ART neural model. *Neural Computing and Applications*, *23*(6), 1815–1829. https://doi.org/10.1007/s00521-012-1147-3

Shi, X., Pruett, S., Doherty, K., Han, J., Petrov, D., Carrig, J., Hugg, J., & Bronson, N. (2020). FlightTracker: Consistency across Read-Optimized Online Stores at Facebook.

Sotiriadis, S., Bessis, N., Amza, C., & Buyya, R. (2019). Elastic Load Balancing for Dynamic Virtual Machine Reconfiguration Based on Vertical and Horizontal Scaling. *IEEE Transactions on Services Computing*, *12*(2), 319–334. https://doi.org/10.1109/TSC.2016.2634024

Sun, C., Sun, D., Agustina & Cai, W. (2018, October 4). *Real Differences between OT and CRDT for Co-Editors*. arXiv: arXiv:1810.02137. https://doi.org/10.48550/arXiv.1810.02137

Van Den Hoogen, I. (2004, January 8). *Deutsch's Fallacies, 10 Years After*. Retrieved March 8, 2023, from https://web.archive.org/web/20070811082651/http://java.sys-con.com/read/38665.htm

Vidot, N., Cart, M., Ferrié, J., & Suleiman, M. (2000). Copies convergence in a distributed real-time collaborative environment, 171–180. https://doi.org/10.1145/358916.358988

Vitillo, R. (2022). *Understanding Distributed Systems* (Version 2.0.0. Second edition). Roberto Vitillo
OCLC: 1348952627.